NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A SOCKETS APPLICATION PROGRAMMING INTERFACE FOR THE PETITE AMATEUR NAVAL SATELLITE

by

Fernando J. Maymí

June 2000

Thesis Advisor:	Bert Lundy
Second Reader:	Jim Horning

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.		
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2000	**3. REPORT TYPE AND DATES COVERED** Master's Thesis
TITLE AND SUBTITLE : A Sockets Application Programming Interface for the Petite Amateur Naval Satellite		**5. FUNDING NUMBERS**
6. AUTHOR(S) Fernando J. Maymí		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.		
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release, distribution is unlimited		**12b. DISTRIBUTION CODE**

ABSTRACT ***(maximum 200 words)***

The Petite Amateur Naval Satellite (PANSAT) is an operational communications microsatellite designed at the Naval Postgraduate School (NPS). PANSAT's communications software was intended to be developed after orbital insertion and transmitted to the satellite.

The Sockets Application Programming Interface (API) developed at the University of California, Berkeley is the de facto standard API for network applications. It provides a strong and flexible platform on which to develop a wide variety of programs. It accelerates the development of new applications by providing a standard set of features and isolating the program from the underlying networking mechanisms.

This thesis studied the viability of implementing of a Sockets API for PANSAT based on the Berkeley Sockets. PANSAT's Sockets API was built on BekTek's Spacecraft Operating System (SCOS). Because SCOS source code was not available, network protocols had to be implemented in user mode. SCOS is optimized for multiple small tasks, not the complex processes required for Internet programming. Because of SCOS' limitations in memory management, the development of this protocol stack and API was not successful. SCOS does not have the features required for an implementation like this.

14. SUBJECT TERMS PANSAT, Internet, TCP/IP, Sockets			**15. NUMBER OF PAGES**
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

A SOCKETS APPLICATION PROGRAMMING INTERFACE FOR THE PETITE AMATEUR NAVAL SATELLITE

Fernando J. Maymí
Major, United States Army
B.S., United States Military Academy, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2000

Author: [illegible]
Fernando J. Maymí

Approved by: [illegible]
Bert Lundy, Thesis Advisor

[illegible]
Jim Horning, Second Reader

[illegible]
Dan Boger, Chairman
Computer Science Department

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Petite Amateur Naval Satellite (PANSAT) is an operational communications microsatellite designed at the Naval Postgraduate School (NPS). PANSAT's communications software was intended to be developed after orbital insertion and transmitted to the satellite.

The Sockets Application Programming Interface (API) developed at the University of California, Berkeley is the de facto standard API for network applications. It provides a strong and flexible platform on which to develop a wide variety of programs. It accelerates the development of new applications by providing a standard set of features and isolating the program from the underlying networking mechanisms.

This thesis studied the viability of implementing of a Sockets API for PANSAT based on the Berkeley Sockets. PANSAT's Sockets API was built on BekTek's Spacecraft Operating System (SCOS). Because SCOS source code was not available, network protocols had to be implemented in user mode. SCOS is optimized for multiple small tasks, not the complex processes required for Internet programming. Because of SCOS' limitations in memory management, the development of this protocol stack and API was not successful. SCOS does not have the features required for an implementation like this.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION

The Petite Amateur Naval Satellite (PANSAT) is a flexible orbital communications laboratory. The Sockets Application Programming Interface (API) developed by the University of California, Berkeley, is a powerful environment for network programming. If the two were to be merged, the resulting system would be flexible and powerful enough to allow for thorough study of the benefits and limitations of simple, low-cost, low-Earth-orbit data communications platforms.

The major obstacle in the development of this system is the limited resources available on PANSAT. The most serious of these limitations, as it turned out, is the way in which PANSAT's operating system manages memory. The system is optimized for the execution of multiple, small tasks, each requiring a very small heap. Though this limitation has little impact on the envisioned Internet applications for PANSAT, it imposes insurmountable obstacles in the development of a user mode Internet Protocol stack. The specific findings are discussed in the following chapters. A fair understanding of PANSAT's architecture, however, is necessary before proceeding.

A. PANSAT OVERVIEW

The Petite Amateur Naval Satellite (PANSAT) is a communications microsatellite designed and built by the Space Systems Academic Group of the Naval Postgraduate School in Monterey, California. It was placed into orbit in October of 1998 by the Space Shuttle Discovery. PANSAT is currently tumbling in a low Earth orbit of about 560 kilometers. On an average day, it offers any given location with a latitude of 36 degrees or less about four communications windows lasting approximately five to ten minutes.

1. Objectives

PANSAT's formal objectives are three-fold:

- to enhance the education of military officers at NPS through the development of a digital communications satellite.
- to serve as a proof-of-concept for a low-cost, spaceborne, spread-spectrum radio communications platform,
- to allow the vast amateur radio community to test and assess the platform, thus demonstrating potential and actual strengths and weaknesses, and

2. Hardware

PANSAT is a microsatellite weighing approximately 150 pounds. It measures 19 inches in diameter and has 26 sides, all but eight of which house solar panels. It has no attitude control or propulsion systems, so it tumbles in orbit. This tumbling characteristic requires the use of an omni-directional antenna to maintain communications with earth stations.

This thesis is primarily concerned with PANSAT's control subsystem. Within this subsystem, I will focus on the central processor, the main storage, and the software that provides the foundation for my development efforts. Though the communications subsystem is fundamental to this effort, its specifics are largely abstracted out by the software environment and will thus be glossed over.

a) *Communications Subsystem*

The spacecraft has a radio transceiver capable of supporting a single half-duplex communications channel. The data rate is dependent on the mode of transmission, of which there are two. This feature makes PANSAT's communications subsystem unique among amateur microsatellites in that it offers narrow band and spread spectrum modes of operation.

The first mode is prevalent among most space and ground radio systems. In narrow band transmissions, the data is encoded on a single carrier frequency. This carrier frequency will then exhibit small variations representing the encoded data. The range of the resulting frequencies, all centered about the original carrier frequency, is quite small or narrow. As a result, if one was to observe the energy of the entire range of radio frequencies at the transmitter's location, one would notice a tall, narrow "spike" centered on the carrier frequency. This feature has been widely exploited by the intelligence community to locate and intercept enemy radio transmissions.

PANSAT's second, spread-spectrum, mode is one in which the transmitted energy is dissipated over a wide frequency range. The total transmitted energy is the same, but it is spread out over a wider range of frequencies. Furthermore, the specific frequencies containing the encoded data are determined using a pseudo-random code which acts as a key to deciphering the transmission. The most interesting consequences of this mode are that the transmitted signal exists below the normal noise level and is extremely difficult to detect, intercept or interfere with and that multiple signals can be encoded on the same portion of the spectrum. This sharing of the spectrum is made possible by the use of different codes for each data channel. The military benefits of spread-spectrum communications should be clear to the reader.

Currently, only narrow band operations have been performed with PANSAT. The mode of operation, though extremely interesting as a research topic, is irrelevant to the work outlined in this thesis.

b) Control Subsystem

The portion of the control subsystem with which this thesis is concerned is the central processor and the main memory. The operating system software, which is the critical limitation in this effort, is discussed separately in the next section.

PANSAT's central processor is an Intel M80C186XL processor operating at 8 MHz. This processor is specifically designed to withstand higher than normal amounts of radiation. This is a necessary characteristic for space operations.

PANSAT's main memory consists of 640 kilobytes of radiation-resistant, self-correcting memory. This is the memory space for all executable tasks. Since a radiation induced error in the task code can lead to a system-wide failure, tasks are required to reside in this protected memory space.

3. Operating System

a) Overview

The BekTek Spacecraft Operating System (SCOS) is a multitasking operating system designed for use on amateur microsatellites such as PANSAT. The primary characteristic that makes SCOS a good microsatellite operating system is that it allows a fairly large number of processes to execute concurrently and communicate with each other. This situation is common in most microsatellites, which have a number of monitoring tasks (such as battery, telemetry, and other sensor controllers) which are fairly small and have to share data with one another.

The most important feature of SCOS, in the context of the design of this Sockets API, is the mechanism for inter-process communications. This mechanism is what would have allowed IP to work on PANSAT without being part of the operating system kernel.

b) Inter-Process Communications

The inter-process message mechanism of SCOS is based on the concept of data streams. SCOS streams are implemented as a first in, first out (FIFO) queue of messages shared among several processes.

In reality each stream is nothing more than a collection of memory buffers within one or more buffer pools. When a process wants to send a message, it simply copies its data to the shared buffer. Any other process with access to this buffer can then read the data by copying it to its own memory space.

Each stream has a unique name that distinguishes it from all others. Stream names are not hierarchical, and thus share a flat namespace. Each process is usually configured with the name of the stream or streams it is to use. There are, however, mechanisms for processes to discover existing streams at runtime.

When a process intends to use a stream, it can create it, check for it and create it if necessary, or just access an existing one. This feature provides an easy method for a task to determine if the underlying services it depends on are running or not. If a task attempts to access the stream of a lower level service and fails, it can assume that the service is not running properly.

In order to send or receive messages over a stream, each process must open a station on the stream. A station is opened with a call to the function **qcf_open**. The function call includes, as a parameter, the name of the stream on which to open the station. The function also allows a process the option to create the stream if it doesn't already exist and then open the station.

When a station is opened on a stream, the process is simply registering with that stream so that it can send and receive messages. It is also initializing the data structures that contain all the information needed to access the stream through the SCOS stream interface functions. It is possible for a process to open a station on a stream, close it, receive messages while absent, reopen the station and read those messages.

Each station on a stream has a control block, where such information as station name, message type, etc. is kept. Before a station can be opened on a stream, the control block values must be initialized. The key field in the control block is the station name, which uniquely identifies the station on the stream. The station name can be chosen by the process or assigned dynamically by SCOS. If the former method is chosen, care must be taken to avoid duplicate names on a stream. It is customary to assign static names only to server tasks, and use dynamically assigned names for clients of those servers. Obviously, a task that is both a client and a server, would need a static name.

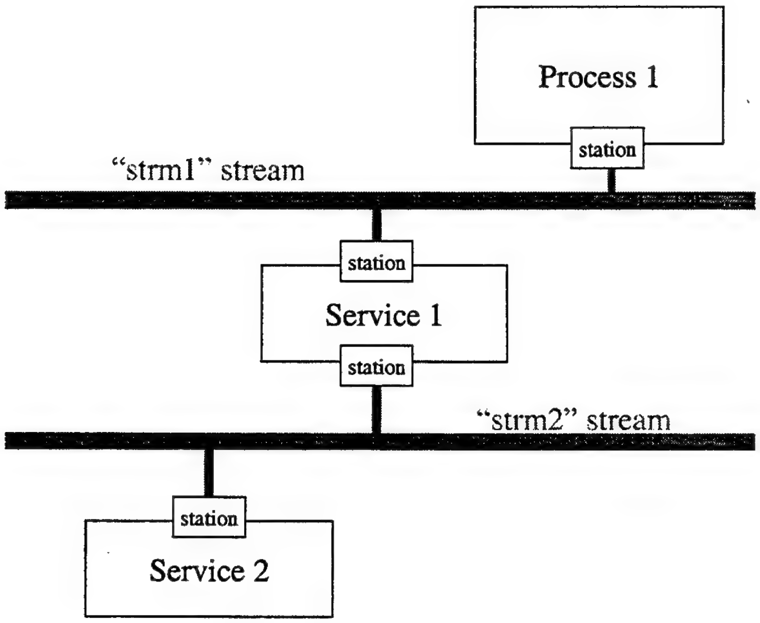


Figure 1 - Generic Streams and Stations

In the figure above, the service Sockets is connected through permanent stations to the "sockets" and "ip" streams. The IP service is only connected to the "ip" stream. Finally, the client applications connect to the transport services through temporary stations that are opened and closed as needed. In this example, which illustrates the implementation of the work on this thesis, the Sockets service is both a server to the client applications, and a client to the IP service.

A message is simply a character buffer with some header information. Specifically, each message has a:

- Source station
- Destination station
- Type (assigned by the source station)
- Data Pointer
- Data Length

When a process receives a message, it actually only gets a pointer to the buffer containing the message. This means that source stations must be careful not to reuse buffers, and destination station must destroy or reuse the buffer after reading the message.

B. THESIS OVERVIEW

This was an effort to determine the feasability of implementing a set of common networking protocols on a particular microsatellite. It was assumed from the beginning that PANSAT is an unlikely platform for this type of software. This assumption was based primarily on the computing and communications resources available on the spacecraft. The combination of a single simplex communications channel with very limited amounts of protected memory makes it extremely difficult to use PANSAT as a spaceborne network server. The benefits of implementing such a system, however, were well worth the risk.

1. Constraints

The major constraints in this effort, were all related to decisions made in the original design of the spacecraft. Specifically, the hardware components and the choice of an operating system constrain this and similar efforts more than any other factor.

a) Hardware

Clearly, all development must be tailored to the available hardware. Due to the difficulties and dangers associated with testing software on the actual satellite, a simulated environment was used for this work. This simulator consisted of a card-mounted processor similar to the one on PANSAT. The simulator also had memory, loaders, and the same operating system as the actual spacecraft. Moreover, the simulator allowed a quick and easy way to test the effects of different configurations without risk to PANSAT.

b) Operating System

The operating system to be used in PANSAT is BekTek's Spacecraft Operating System (SCOS). SCOS was introduced in the preceeding section. This constraint is significant in part because SCOS is proprietary and its source code is not available to developers.

What this means to the development of a network protocol stack is that many of the features that would normally be available in the kernel are off-limits in this case. All common protocol stacks are implemented as part of the kernel of whatever operating system they support. Kernel mode processes have special privileges and permissions that are not available to user mode processes.

In order to develop this IP stack, all services have to be implemented as user mode processes. This, in and of itself, significantly increases both the complexity of the software and the amount of memory required by it. These, as you will see in the following subsection, are two of the factors I strove to minimize in developing this system.

c) Link Layer

The link layer that had to be used for this project is that provided by BekTek's implementation of the AX.25 protocol, known as BAX. As it turned out, this was one of the major obstacles in achieving compliance with Internet standards. BAX imposes a limit of 256 bytes on the data portion of a transmitted frame. This is well below the requirement for IP's minimum transmission unit (MTU), which is 576 bytes.

2. Requirements

This project was born from a desire to implement a standards-compliant networking protocol on PANSAT. As a result, the requirements are clearly stipulated by various publications. It soon became clear, however, that full compliance with these standards would be extremely difficult, given the existing constraints.

a) Internet Protocol

The requirements for hosts implementing the Internet Protocol are listed and discussed in Request for Comments (RFC) 1122 and 1123. The protocol itself is discussed in RFC 791. The specific requirements are listed in Appendix A.

b) Transport Protocols

The transport layer protocols that are normally part of an IP stack are the User Datagram Protocol (UDP) and the Transport Control Protocol (TCP). Early in the development of this effort, TCP was eliminated from the implementation list because of the amount of memory this would require. Since the purpose of this thesis was to determine the feasibility of implementing the IP stack, I decided to keep the scope as small as possible to improve the chances of success. The list of requirements for the UDP protocol are listed in RFC 768.

c) Sockets Interface

Though there is no standard document that specifies the requirements of a Sockets Application Programming Interface (API), there are numerous publications describing what has become the de facto standard Sockets API: the Berkeley Sockets developed by the Berkeley Software Development (BSD) at the University of California, Berkeley. The most common features of this interface are listed in the **sockapi.h** file, which is listed in Appendix E.

II. BACKGROUND ON PREVIOUS WORK

George Kenneth Hunter provided a design for PANSAT's user-services software in his Masters thesis in 1998. His design is for an electronic bulletin-board system similar to those that preceeded the Internet in the 1970s. This design is prevalent among the amateur radio community.

More recently, Michael Adelhardt of the german Universitet der Bundeswehr Hamburg, improved upon and implemented Ken Hunter's design while working at NPS. His work, while complete, has not yet been tested on the spacecraft, because the file system is not yet in place.

While the use of TCP/IP on amateur radio networks is clearly increasing, most implementations remain proprietary and their source code is not generally available. The support provided by the Linux operating system for the AX.25 protocol is likely to fuel this trend. At this time, however, TCP/IP is not widely used by the amateur radio community.

THIS PAGE INTENTIONALLY LEFT BLANK

III. OVERVIEW OF THE PANSAT SOCKETS

This is a minimal implementation of the Internet Protocol (IP), Internet Control Message Protocol (ICMP), and User Datagram Protocol (UDP) that is accessed through an application programming interface (API) that is almost identical to the Berkeley Sockets API. Due to platform and operational considerations described in Chapter I, only the features required by the appropriate requests for comments (RFCs) are implemented. The primary design objective supported by this minimal implementation is to minimize the onboard resources (memory, CPU cycles) required of PANSAT.

PANSAT is assumed to always act as a network host (as opposed to a router). As such, no routing mechanisms are part of this implementation. However, since the underlying link layer protocol, BekTek's AX.25 (BAX), supports multiple addresses, PANSAT could be configured with multiple virtual interfaces. Though the interfaces would be distinct, they would not be allowed to route packets to each other. In this scenario, each interface requires a unique IP address and PANSAT would be "mulihomed." Multihoming, though supported by this implementation, is considered beyond the scope of this work and is not included in the test plan. Note, however, that PANSAT is configured, by default, with two interfaces: a BAX network interface and a loopback interface.

A. INTRODUCTION TO SOCKETS

The Sockets API developed by the University of California at Berkeley is based on a model of network communications wherein each networked host needs one or more "sockets" or end connections on a communications medium in order to exchange messages with other, similarly configured, hosts.

A socket, then must be characterized by the medium on which it exists. This medium is a specific protocol (e.g. the User Datagram Protocol or UDP) within a protocol family (e.g. the Internet Protocol or IP). This means that a socket can only understand that protocol for which it was created. A socket's protocol cannot be changed.

When a socket is created, it has no network identity. That is to say that the socket can not receive any messages. If an "anonymous" socket such as this attempts to send a message to another socket on the network, the Sockets layer automatically assigns the socket a temporary identity. This identity is discarded upon conclusion of the transmission. If PANSAT supported the Transport Control Protocol (TCP), which is a connection-oriented protocol, an "anonymous" TCP socket would not be able to transmit messages at all.

The identity of a socket is defined by three parameters: address, protocol and port number. In PANSAT, the address will always be an Internet address, since IP is the only network protocol supported. The protocol can be either UDP or Raw IP. The port number can be assigned by the application for the life of the socket, or it can be dynamically assigned by the Sockets layer for the duration of a single transmission. The first method is used by a server process which must have a well-known (to other hosts) port number so that others can always connect to it. The second method would not be used by any conceivable application on PANSAT, since it presumes a single message is transmitted with no possibility of receiving a response.

After a socket's identity has been specified, that socket may transmit and receive messages. If the socket is on a server application's well-known port, all connection requests for that server will arrive at that socket. If multiple simultaneous client connections are desired, the socket can be made to spawn new sockets, one for each new connection, so that the well-known socket is not monopolized by any one client.

B. SOCKETS ON PANSAT

Sockets are normally implemented as part of the operating system through the use of system calls. Because PANSAT uses BekTek's Spacecraft Operating System (SCOS), and we do not have access to its source code, the sockets API is implemented through the use of library functions. These functions send predefined messages (corresponding to system calls) to the transport services (UDP or Raw IP) over SCOS streams. These messages have a one to one correspondence to the function calls of a kernel-based sockets implementation. They are a good method of providing the same sockets interface found on other operating systems.

To minimize the overhead required by this implementation, the Sockets and transport layers were merged. The result is the architecture depicted in the illustration. The dashed lines indicate entities which are dynamically created when needed and destroyed afterward.

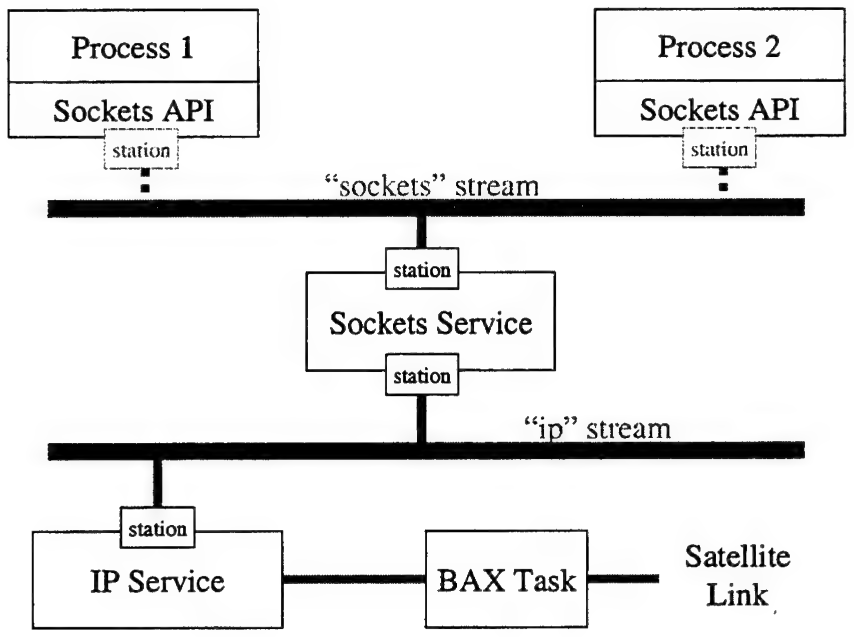


Figure 2 - Protocol Stack Architecture

Applications are linked to the Sockets API at compile time. During execution, the application calls the socket functions as it would any other. The API handles all communications with the underlying Sockets and IP layers.

All messages generated by the Sockets API for the Sockets service start with a standard header. This header consists of a **sock_msg_hdr** structure, which has the following fields.

```
Struct sock_msg_hdr {
    int    sock_id;
    short  sys_call;
    short  rtn_code;
};
```

What follows this standard header depends on the specific Sockets system call identified in the **sys_call** field. The following table contains the structure of the encoding of all the currently defined system calls in the PANSAT Sockets API.

System Call	1st Structure	2nd Structure
SOCKET	sock_msg_hdr	param_socket
BIND	sock_msg_hdr	sockaddr_in
LISTEN	sock_msg_hdr	int
CONNECT	sock_msg_hdr	sockaddr_in
ACCEPT	sock_msg_hdr	n/a
SEND	sock_msg_hdr	param_dgram
SENDTO	sock_msg_hdr	param_dgram
RECV	sock_msg_hdr	param_dgram
RECVFROM	sock_msg_hdr	param_dgram
CLOSESOCKET	sock_msg_hdr	n/a
SHUTDOWN	sock_msg_hdr	int

Table 1 - System Call Structures

For instance, when an application calls **socket**, it is actually sending a message buffer containing a sock_msg_hdr structure followed by a param_socket structure. Both structures' values are set by the **socket** function. If the actual call was **socket(AF_INET, SOCK_DGRAM, 0)**, then the transmitted message would be as indicated in the figure below. (Notice that the SOCKET code is defined with the value 1, and AF_INET and SOCK_DGRAM are both constants defined with the value 2.)

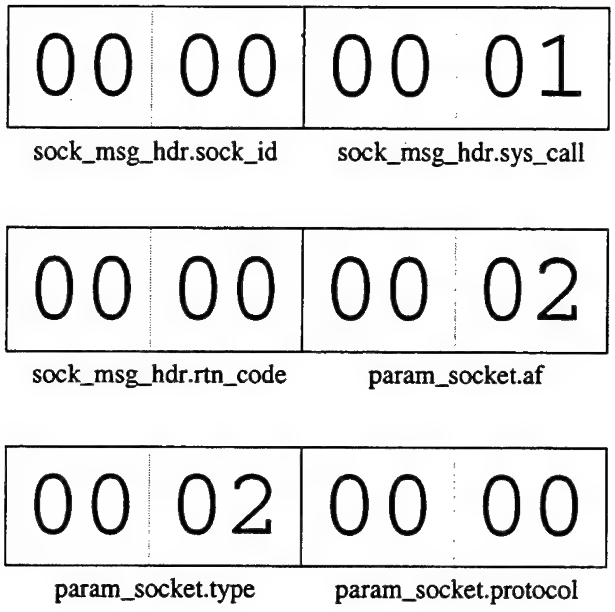


Figure 3 - SOCKET Call Encoding

When a sockets message is sent to the transport service, the sockets API function will block until a response is received from the transport layer. When a transport layer service receives a message from the Sockets API, it processes the message and returns a result/error code. At this point, the Socket function will return the code to the calling process. Since all sockets data is kept by the Sockets service, it is possible to develop non-blocking Sockets calls. This feature, however, is not part of this implementation.

The Sockets service maintains all socket data and handles all socket actions to include sending and receiving messages. The socket data structures maintained by the Sockets service include input and output message queues to support non-blocking sockets.

The Sockets service sends and receives messages from the Internet Protocol (IP) service below it through SCOS stream messages. The sockets service executes a cycle of checking for messages from the applications and from the IP layer. The cycle also includes basic housekeeping functions.

C. EXAMPLE APPLICATION

The following application example will serve to illustrate the operation of the PANSAT Sockets API and the underlying transport, network, and link layer protocols. The application is a simple time server, which sends each client a UDP datagram with the current PANSAT system time.

1. System Initialization

When the IP module is loaded into PANSAT, it begins execution immediately. First, the IP module creates a SCOS stream with which to communicate with the transport services, and creates a station for itself on that stream.

This allows the transport modules to initialize, since they would fail initialization if they cannot find the IP stream.

The IP module then attaches all defined interfaces on the network (link layer) side. The first interface initialized is the loopback interface. Then any other interfaces defined are initialized by claiming callsigns for them with the BAX task. This step allows other hosts to establish AX.25 connections to the IP interfaces through the BAX task.

At this point, the IP module is initialized and ready to provide services. The module goes into an endless loop of checking for input (from the interfaces), then checking for output (from the transport services). The IP layer then executes any required administrative tasks before starting the loop again.

After the IP module is loaded and executing, the Sockets module can be loaded and executed. When the Sockets service begins execution, it first initializes the data structures it will need to support the applications. The Sockets service then looks for the sockets SCOS stream with which it will communicate with the Sockets API. If the stream exists, the service opens a station for itself on that stream. If the stream does not exist, the service will first create it and then open a station for itself.

At this point, the Sockets service module is initialized and ready to provide services. The module goes into an endless loop of checking for input from the "sockets" stream, then checking for input from the "ip" stream. The Sockets service then executes any required administrative tasks before starting the loop again.

2. Application Initialization

Now, a sockets application can be loaded and executed. The example application is a time server. First, the process will open a socket through the **socket** function call, passing AF_INET (IP) as the address family parameter, and SOCK_DGRAM (UDP) as the protocol parameter. The **socket** function opens a station on the "sockets" stream and sends a message to the Sockets service requesting a new socket. The Sockets service allocates a new structure for the socket, assigns it a unique id and sends a message back to the application confirming the creation of the socket and providing the unique identifier for it. Once the **socket** function receives this message from the Sockets service, it closes its station on the sockets stream, and returns the identifier, or handle, to the calling process.

The time server now has a valid socket to initialize. The server application next calls the **bind** function to provide the unique network identity of the socket to the Sockets layer. The application provides the IP address for the socket (optional, since IP can do this for each outgoing datagram), and the port number (required, since it must be a well-known port to other Internet hosts). If the IP address is not set by the application, its value is 0.0.0.0, which signals the IP service to assign any outgoing messages the IP address of a valid interface. This assignment does not affect the socket, only the outgoing datagram. The **bind** function opens a new station on the sockets stream and sends a BIND request together with the provided parameters to the Sockets service. The Sockets service sets the address and port values in the socket structure and sends a confirmation message with a result code to the **bind** station. Once **bind** receives this result, it closes its station and returns the code to the calling process.

Now, the server application configures the socket to accept incoming connections. To do this, it calls the **listen** function and provides, as a parameter, the length of the queue for incoming connection requests. The **listen** function opens a station on the sockets stream and sends the request and parameter to the Sockets service. The Sockets service sets the flag on the socket structure that enables it to accept connection requests, sets the maximum length of the pending queue, and sends back a result code to the **listen** function. Once **listen** receives this message, it closes its station on the sockets stream and returns the code to the calling process.

The last step in the setup of the time server application, is for it to wait for incoming requests from client hosts. It does this by calling the **accept** function, which again opens a station on the sockets stream, sends an ACCEPT message to the Sockets service, and waits indefinitely for a new connection. The Sockets service will set the flag on the socket indicating that it is waiting to accept a connection, and continues to execute. When the IP module sends an incoming datagram destined for this waiting socket, the Sockets service will notice the state of the socket and finish executing the ACCEPT.

3. Serving a Client's Request

When a client wants to get PANSAT's system time, it first establishes a connection with PANSAT over AX.25. The connection request includes the callsign of one of the IP service's interfaces on PANSAT. The BAX task receives the connection request from the client and searches for a task with a matching callsign. In this case, it should find the IP service (assuming it succesfully initialized at least one network interface). BAX then forwards the request for an AX.25 connection to the IP task.

When IP checks for pending input on its interfaces, the recipient interface will receive the connection request. Assuming the interface has not been configured to refuse connections, it will automatically send a control packet accepting the connection through the BAX task to the client system. The interface status flag will indicate an active connection.

When the client receives the connection acceptance message from the IP interface, it transmits the AX.25 data message containing the IP datagram (which, in turn, contains the UDP datagram to the time server socket). When the BAX task receives the data message, it stores it in the buffer for the IP task.

The next time the IP process checks its interfaces for pending incoming traffic, the interface will query the BAX task and receive the message from the client. The IP module will process the datagram it received, determine that it is intended for the Sockets service. The IP module will then put the datagram on the "ip" SCOS stream for the "sockets" station.

The Sockets service will check for new messages for it on the "ip" SCOS stream, receive the datagram, and match it to the time server's socket. Since the socket is set to accept connections, the Sockets service will create a new socket, place the received datagram in that socket's input queue, clear the accept flag from the original socket (until another call to accept sets it again), and sends a message to the waiting process with the handle for the newly created socket where the datagram can be found.

The time server's call to the **accept** function finally returns the handle for the new socket. Since the contents of the client request datagram are irrelevant, the time server does not need to receive it. Instead, the server will send the current system time to the distant end of the newly created socket (the client), by calling the function **send** for the new socket.

When the server calls **send**, that function will create a new station on the "sockets" stream, send a message (which includes the data for the outgoing datagram) to the "sockets" station, and wait for acknowledgement. The Sockets service receives the message from the time server, generates a UDP header for it, prepends a generic IP header on the whole packet, and sends the partially initialized IP datagram to the IP service via a SCOS stream message on the "ip" stream. At this point, the UDP service replies to the **send** message from the time server with a succesful result code. Note that the Sockets service does not verify (nor does it care) that the datagram made its way to the recipient. Upon receiving the SEND acknowledgement, the time server's **send** function closes the station on the "sockets" stream, and returns control to the time server application.

Meanwhile, the IP service validates the addresses, completes the IP header, and sends the datagram out over the corresponding interface. Any errors encountered during the transmission are reported asynchronously to the Sockets service via an error message on the "ip" stream. The Sockets service would receive the message and update the data on the socket's structure to reflect the appropriate error code.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. AX.25

A. OVERVIEW

AX.25 is a non-standard variant of the X.25 protocol. AX.25 is adapted for use in packet switched networks over amateur radio links. The protocol, though widely used by amateur radio enthusiasts world-wide, is not currently defined in any standard.

The implementation of AX.25 that is part of SCOS is BekTek AX.25 (BAX).

B. BEKTEK'S AX.25

1. Data Structures

The fundamental data structure of BAX is the Control Block. Defined in **struct CONTROL_BLOCK** in the qax25.h file, this structure contains the various information fields relevant to a given packet. The fields are set by the **qax_input** function. The fields can also be manually set to describe an outgoing packet, such as that used for a connection request by the **qax_connect** function. In this situation, any irrelevant fields are ignored by the BAX task. The structure is given below.

```
struct CONTROL_BLOCK {
   unsigned int channel;      /* recvd packet channel */
   unsigned int type;      /* returned packet type */
   unsigned int state;     /* channel state variable */
   unsigned int cause;     /* reason for state change */
   unsigned int outstanding;  /* internal use only */
   unsigned int loststate;    /* internal use only */
   unsigned char modem;    /* internal use only */
   struct AX25_ADDR my_call;  /* this station callsign */
   struct AX25_ADDR his_call; /* other station callsign */
   unsigned char pid;      /* PID value for I and UI */
   unsigned char t1;       /* timeout value */
   unsigned char maxframe;    /* max frames in flight */
   unsigned char retry;    /* max retries */
   unsigned int paclen;    /* max size of sent packet */
   unsigned char rx_chan;     /* received port number */
   unsigned char rxs;      /* not used */
   unsigned char rxd;      /* not used */
   struct AX25_ADDR path[MAXDIGIS]; /* path to follow */
};
```

2. External Interfaces

BAX's external interfaces are defined in the file qax25.h. The functions that provide this interface can be divided into four categories: interface, input, output, and control.

The BAX interface functions listed in Table 2 allow an application to register or release a callsign with the BAX task. This is an essential step, as callsigns are the node addresses for AX.25, and without them no process may transmit or receive packets.

Function	**Description**
qax_claim	Informs BAX that the named callsign is handled by this application.
qax_return	Informs BAX that this application in no longer handling the named callsign.

Table 2 - BAX Interface Functions

The input functions listed below read data or control packets from a channel.

Function	**Description**
qax_input	Waits for input.
qax_input_ready	Checks for pending input.
qax_last_time	Returns the time that the last frame returned by qax_input was received by the BAX task.

Table 3 - BAX Data Input Functions

The data output functions listed in the following table allow applications to transmit AX.25 packets.

Function	**Description**
qax_data	Sends data.
qax_data_full	Checks for room in data output queue.
qax_ui	Sends a UI frame.
qax_ui_full	Checks for room in the UI data output queue.

Table 4 - BAX Data Output Functions

The functions listed in the following table cause control packets to be sent through a channel.

Function	**Description**
qax_busy	Sets the channel state to busy, all received I frames will be acknowledged with an RNR.
qax_con_acpt	Accepts a connection.
qax_con_rej	Rejects a connection.
qax_connect	Sends a connection request.
qax_clean_cb	Clears a control block.
qax_disconnect	Disconnects.
qax_unbusy	Clears the state set by **qax_busy**.

Table 5 - BAX Control Functions

There is an additional BAX function which doesn't fit nicely in any of the above categories. This function is **qax_stats**, which provides a wide range of statistics maintained by the BAX task.

V. THE INTERNET PROTOCOL

A. OVERVIEW

The Internet Protocol (IP) is defined in RFC 791 (The Internet Protocol). The requirements for hosts implementing this protocol are listed and explained in RFCs 1122 and 1123 (Hosts Requirements).

The figure below illustrates the structure of an IP datagram. The grey blocks indicate fields that are normally set by the application via the Sockets layer. The identification, flags, and fragment offset fields are used by IP to handle fragmented datagrams. Source and destination addresses, though assigned by the application, are verified (and modified if need be) by the IP service. The payload area of the datagram can contain the header information for a higher layer protocol, such as UDP.

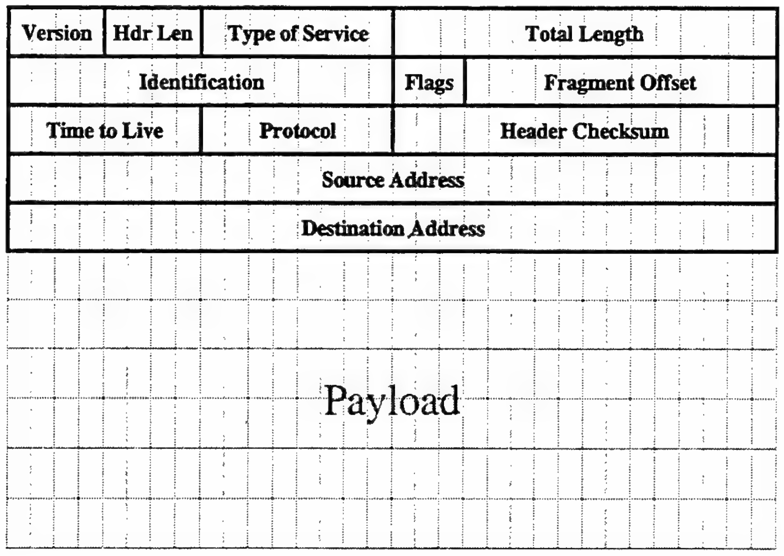


Figure 4 - IP Datagram

PANSAT's implementation of IP was to be minimal, yet sufficient to satisfy most of the requirements of the applicable standards. PANSAT's IP is divided logically into three modules: interfaces, IP, and ICMP. Each module was implemented as a separately compiled file. The dependencies between these modules are depicted in the following figure. As can be seen, SCOS provides the underlying services used by all modules, primarily in the area of dynamic memory management. Dynamic memory management was critical in this design, since the heap allocated to each process was fairly small, and SCOS allowed access to kernel memory areas through the allocation of FAR pointers.

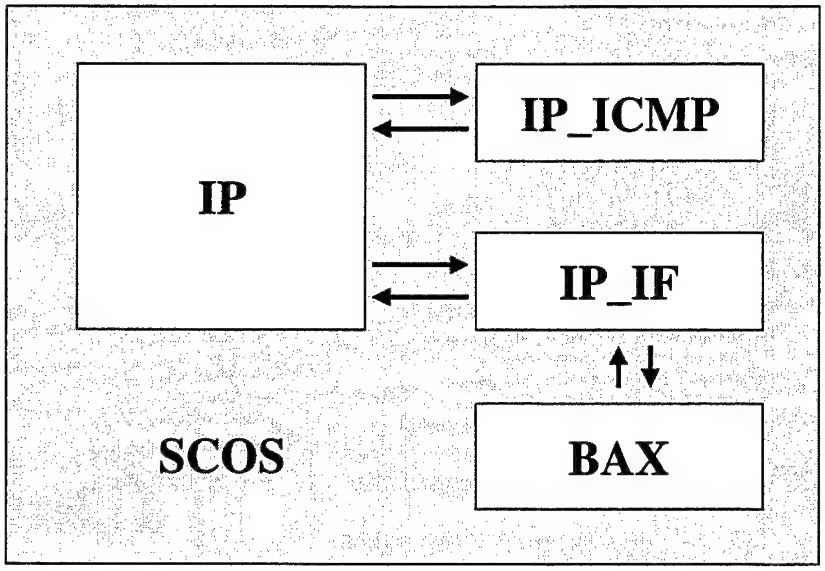


Figure 5 - IP Module Dependencies

An important characteristic of the system is that the interface module (ip_if) isolates the rest of the stack from the BAX task. This was done intentionally to allow for the possibility of later circumventing BAX and using a different link layer mechanism that would support larger frames than BAX's 256 byte limit.

B. THE INTERFACE MODULE

1. Overview

This module handles all the details of interfacing with the link layer (BAX). This is important to relieve the IP module from the complexities associated with opening, managing and closing AX.25 connections. It also allows all interface (BAX, loopback, plus any future ones) implementation details to be transparent to the IP module.

2. External Interfaces

The external interfaces to the interface module are defined in the file ip_if.h. The functions are listed and explained below. Though they are external to the separately compiled interface module, they are to be used only by the IP service.

a) if_attach()

The **if_attach** function receives as arguments an address and netmask for the new interface, as well as control block information and a pointer to the list of interfaces. It returns a pointer to the new list of interfaces with the new interface prepended to it. New interfaces are always added to the head of the interface list.

If the control block argument is not null, **if_attach** registers the interface with the BAX task, so that it may transmit and receive messages. The control block would be null if the interface is a loopback interface and need not be registered.

b) if_detach()

If an interface is not longer needed, it may be removed from the system using this function. The interface to be removed need not be at the head of the interface list. The function returns a pointer to the new list of interfaces with the selected one removed.

c) if_disconnect()

If an interface is in a connected AX.25 state, this function can be called to terminate the connection. This is useful during regular housekeeping maintenance to prune stale connections.

d) if_find()

The **if_find** function searches for an interface that matches the address and/or netmask provided as arguments. The function is used to determine the interface to be used in transmitting a particular datagram. Since the interface to use depends on the netmask of the destination, this function allows the IP module to support multiple interfaces.

e) if_send()

The **if_send** function places a message on the outgoing queue of the given interface and attempts to flush the queue. Messages are processed in the order in which they were received by the interface, so a failure to flush means that the message remains in the queue until the next flush attempt is successful. The function returns an error if the interface is not connected.

f) if_receive()

To check for new messages on an interface, the function **if_receive** is called. The function will first query the BAX task for new pending messages. If any are found, as many of them as will fit are placed in the interface's input queue. If there are more pending messages than room for them in the queue, the messages that do not fit are not received from the BAX task. The function **if_receive** will then examine the interface's input queue and return the first message in it. Only one message is returned for each call to this function.

3. Internal Design

The interface module is characterized by a single structure (**struct ip_if**) and a number of function calls to handle instances of this structure. All instantiated ip_if structures are kept in a linked list by the IP module. When a new ip_if structure is created via the attach function, it is inserted into the head of the list of ip_if structures. The attach function takes a pointer to the existing list (if any) and returns a pointer to this new list with the new interface at its head. This methodology was chosen because the loopback interface will be the first to be attached, but should be the last to be checked when searching the interface list. This approach also simplifies the process of adding interfaces and provides us a way to speed up the search process by attaching the most commonly used interface last.

Before the interface is attached to the system and ready for use, the attach function initializes a BAX control block for the interface and claims a callsign. When attach returns, the interface is ready to accept a connection, but cannot send or receive packets until a connection has been established. AX.25 connections are accepted by the interface module whenever an interface is checked (polled) for incoming traffic. If the incoming message is a connection request, the interface will automatically complete the connection unless it has been explicitly configured to refuse additional connections. Likewise, inactive connections are checked every time the system checks for new traffic and are terminated after their timer has expired.

Each interface has an associated input and output queue. The behavior of these queues can be controlled through the use of the flags available in the ip_if structure. They can be configured, for instance, to refuse incoming messages while continuing to transmit over the connection. They can also be made to transmit messages immediately rather than queueing them for transmission. These flags, as well as all other interface characteristics, are controlled through the **if_configure** function. This function takes as an argument a value for the interface status flag and sets that flag. The same result could be obtained by directly setting the value, but the approach chosen encourages access to the data through predefined functions in a manner similar to that used in object-oriented programming.

When a message is received or ready to be transmitted, the appropriate interface is found through the **if_find** function. This function will match an interface to a source or destination address or netmask or both. When the right interface has been found, the interface module checks that interface's in- or outbound queue. If there are any messages already waiting to be processed, these will be handled first. If the message is incoming, and there are other incoming messages waiting in the queue, the new message is appended to the list and the oldest message is returned by the **if_send** function. If the message is outgoing, and there are other messages waiting to be sent, the interface will attempt to flush all outstanding messages first. Then it will attempt to transmit the new message.

If the the outbound queue is full and the BAX task cannot accept any more messages from the IP service, **if_send** returns an error code to the IP module. If the inbound queue is full, **if_receive** will not accept any more messages from the BAX task until the queue has room.

C. THE IP MODULE

1. Overview

The IP module consists of a number of sub-modules. There is an input module which checks for and processes messages arriving through one of the interfaces. This module has a specialized function to handle datagram reassembly. It is also coupled with the ICMP module (described later) to send out error messages when a bad datagram is received.

There is also an output module that checks for and processes messages arriving through a SCOS stream.

2. External Interfaces

The interface into the IP service is defined by the SCOS stream functions and the BAX functions. The IP service interprets every received message as an IP datagram. If the received message does not pass the standard tests for IP datagram validity (header length, version, checksum, etc.), the message is discarded and may generate an ICMP message.

BAX connection requests are handled by the interface module, not by the IP module. In this way, the design maintains the consistency required to treat every incoming message to the IP module as an IP datagram. All the overhead related to the AX.25 protocol is hidden from IP by the interface module.

3. Internal Design

a) Common Function Calls

The IP module includes two helper functions that are available (through the ip.h file) to other layers. These are chksum() and charcopy().

(1) chksum() - This function performs a very simple Internet checksum of a given character buffer. Though this buffer is usually an IP header, it can also be used for UDP headers or other purposes.

(2) charcopy() - This function copies a given number of characters from the given source and source offset to the given destination starting at the given offset. This function is useful for encapsulating messages of a higher protocol in preparation for transmission, and for extracting messages from lower protocols as they travel up the protocol stack.

b) Input Processing

IP checks for new inputs through the **ip_input** function. This function sequentially checks all attached interfaces for messages in their input queue. If any are found, they are processed in the order in which they were received.

When a new message arrives through one of the interfaces, the IP module will first verify its validity by checking such parameters as IP version number, source or destination address, checksum, etc. It then determines whether the message is a complete datagram or a fragment of a datagram. If it is a fragment, the **ip_reassemble** function first checks to see if all fragments have arrived. If they have, the function will reassemble them. Otherwise, the fragment is inserted into the appropriate fragment list to wait for the other fragments of the datagram.

When a datagram fragment is being processed, the **ip_reassemble** function will only keep the data it needs. Specifically, only one copy of the datagram header is maintained per list of fragments. This data is stored in a special structure at the head of the list for the particular fragmented datagram. Beneath this structure lies a list of fragment buffers. These structures only have a fragment's portion of the datagram payload, plus start and finish offsets for this data within the completed datagram. This method of storing the datagram fragments saves space and facilitates the reassembly of the datagram once all fragments have been received.

If the incoming datagram is still being processed, that is, if it has not been discarded as bad or inserted into a fragment list, the IP module uses the protocol field to demultiplex it. If the protocol is ICMP, the message is passed to the ICMP module through the **icmp_input** function. The specifics of the ICMP module are discussed in the next section. If the protocol is other than ICMP, the IP module will pass the datagram on the "sockets" stream and send it to the Sockets service. If the protocol is other than UDP or Raw IP, or if the Sockets service is not responding, an ICMP message is generated through the **icmp_output** function. Notice that incoming messages are not buffered by the IP module after being processed. Instead, they are immediately sent to the corresponding protocol service via a SCOS stream.

c) Output Processing

The IP module checks for new messages from the transport layer through the **ip_output** function. This function checks the SCOS stream for pending messages. If any are found, the function fills in the appropriate fields in the IP header, such as version, header length, identification, etc.

ip_output also validates the source and destination addresses. The source address must match the address of one of the attached interfaces. If this is not so, the value of the source address is modified to satisfy this requirement. The destination address, on the other hand, is only verified for conformance with the structure of class A, B, and C addresses. IP broadcast packages are not supported by PANSAT's IP module. Correct addresses of each class are illustrated in the table below.

Address Class	**Smallest Value**	**Largest Value**
A	1.0.0.1	126.255.255.254
B	128.0.0.1	191.255.255.254
C	192.0.0.1	213.255.255.254

Note that there is a special case of a Class A address which is a loopback address. This loopback is defined as any address whose first octet is 127. Addresses whose host part is all zeroes represent networks rather than hosts and are not allowed for transmission. Similarly, addresses whose host part is all ones represent broadcast addresses for the appropriate network and are not allowed to be the source or destination of an IP datagram.

After all checks and required changes are made, **ip_output** calls the **if_send** to transmit the datagram. The only condition which would not allow a datagram to be sent is a bad destination address. All other corrections are handled dynamically as the outbound datagram is being processed.

D. THE ICMP MODULE

1. Overview

The Internet Control Message Protocol is defined in RFC 792. It is primarily a protocol for the exchange of status messages between network hosts. It is a required feature of an IP implementation.

The most common use of ICMP is in determining when a host is unreachable. Recall that the IP module can only check an outbound address for conformance with the structure of the corresponding address class. IP has no way of knowing whether the host exists or is connected to a network. If the host were unreachable, a router on the destination network will determine that the datagram cannot be delivered. The router will then discard the datagram and generate an ICMP message notifying PANSAT that this is a bad destination address. PANSAT's ICMP module will then notify the Sockets module of this fact.

Another common use of ICMP is to request and respond to echoes. An echo request is essentially a message telling the receiver: "If you are there, respond!" ICMP echo requests, also known as "Pings," are among the most useful tools in troubleshooting network connectivity.

PANSAT's ICMP implementation supports a subset of the features of a complete system. PANSAT's ICMP, however, satisfies all requirements of the standard for the protocol. The key services provided by ICMP on PANSAT are passing datagram error messages (destination port unknown, etc.) and operating an echo server. These are essential for the anticipated usage of the satellite.

2. External Interfaces

ICMP is called by the IP module via two functions: **icmp_input** and **icmp_output**. The first handles incoming datagrams whose protocol identifier is ICMP. The second function is intended for sending ICMP messages that have been generated by PANSAT, whether in response to a received bad datagram or in response to an external ICMP request (echo). Notice that ICMP messages generated by applications through the Raw IP service are not handled by the ICMP sub-module. This means that if, for instance, an application sends out an ICMP echo request through the Raw IP service, IP will simply forward the datagram, untouched, to the appropriate interface for transmission.

a) icmp_input()

When IP receives an ICMP message through the **ip_input** function, it calls the **icmp_input** function to handle it. The **icmp_input** function examines the message type and generates the appropriate response. Currently, only two types of ICMP messages are handled by ICMP: Echo Requests and Destination Unreachable error messages.

If an ICMP echo request is received, **icmp_input** generates an echo reply message and calls the **ip_output** directly to send it. If an ICMP error message is received, the function examines the IP header of the datagram that caused the error message to determine which transport protocol service should receive the message. All ICMP error messages must include at least the first 28 bytes of the datagram that caused the message to be generated.

b) icmp_output()

When the IP module cannot deliver a received datagram, it calls **icmp_output** to generate an ICMP error message and transmit it to the datagram's source. **icmp_input** receives the bad datagram and ICMP codes from the IP module, generates the proper ICMP error message, and calls **ip_output** directly to transmit the ICMP message.

E. CONCLUSION

The implementation of IP on PANSAT is sufficient to perform the most common tasks required of an Internet host in general and of PANSAT in particular. Though not all required features are supported, this work represents a good foundation on which to experiment and build. The goal of maintaining code size and efficiency, as well as operating system overhead, down to a minimum has been accomplished by this system.

It is important to remember that many features that Internet applications take for granted are not supported yet. The effects of these limitations on actual service characteristics should be studied to determine what changes should be made to the IP service on PANSAT.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. THE TRANSPORT PROTOCOLS

A. OVERVIEW

Because of the requirement for an extremely small implementation of IP on PANSAT, the only transport protocol that is part of this system is the User Datagram Protocol (UDP). After sufficient tests and analysis have been conducted, other protocols (specifically the Transport Control Protocol or TCP) could be added.

In addition to UDP sockets, support for Raw IP sockets is provided as a transport layer service. The Raw IP service, however, is not a transport protocol, as all issues normally related to the transport layer are handled by the application layer when it uses Raw IP sockets.

B. SOCKETS INFORMATION

All data pertaining to the sockets is maintained by each transport service in a doubly-linked list of socket structures. These structures contain all information required to operate the sockets, including their identifiers, and message queues. The socket structure is shwon below.

```
struct socket {
   int          handle;
   unsigned long  sock_address;
   unsigned short sock_port;
   unsigned long  dest_address;
   unsigned short dest_port;
   short        type;
   short        options;
   short        flags;
   short        state;
   char far     *in_q[QUEUE_SIZE];
   char far     *out_q[QUEUE_SIZE];
   struct socket  *next_socket;
   struct socket  *prev_socket;
};
```

The socket handle is a unique identifier for the socket. This value is stored in a variable of type integer. This means that no two sockets, regardless of type, may ever have the same handle. To satisfy this requirement, the Sockets service maintains a global integer variable which is incremented as sockets are created. Though this means that socket handles are not recycled, it should not be significant in PANSAT. The satellite, as was stated in Chapter I, has very limited memory resources. This implies that the maximum number of over 65 thousand sockets will not be reached in the lifetime of the satellite. To do this would require the satellite to remain operational, without resetting, for all six years of its estimated orbital lifetime. Additionally, each of those days, 45 sockets must be created. Considering that, due to antenna damage during orbital insertion, only the ground station at NPS can communicate with the PANSAT, these numbers are extremely unlikely to be reached.

C. THE USER DATAGRAM PROTOCOL

The User Datagram Protocol (UDP) is defined in RFC 768. UDP is an almost null protocol. The header only provides source and destination port numbers, header length and a header checksum. All error and flow control is the responsibility of the application using the protocol.

UDP is the ideal transport protocol for this initial implementation of IP on PANSAT. It is combination of extremely low overhead and support for multiple application protocols makes it the right choice for this testbed. The specific protocols that UDP will probably support on PANSAT are the Trivial File Transfer Protocol (TFTP) and the Network News Transfer Protocol (NNTP). TFTP will be useful in uploading and downloading files to the satellite. NNTP will provide a simple, inexpensive means of supporting PANSAT's mission as a "bulletin board in the sky." TFTP and NNTP, though interesting and useful for PANSAT, are beyond the scope of this work. The test application will be a UDP time server.

D. THE RAW IP SERVICE

The Raw IP service provides a means for applications to generate their own IP datagrams without any IP layer intervention. This is useful when writing applications such as the Ping program, which generates ICMP echo requests to determine if a given host is reachable over the network.

When the Raw IP service is used, the Sockets service simply acts as a relay agent to send the datagrams created by the application directly to the IP module. The Sockets service does not alter these messages at all. Similarly, the IP service performs only minimal processing of these datagrams. Other than verifying addresses, the IP service does nothing to these messages before relaying them.

VII. THE SOCKETS APPLICATION PROGRAMMING INTERFACE

A. OVERVIEW

The Sockets API allows programmers to ignore the internal workings of the network protocols stack on PANSAT. The interfaces defined by the Sockets API allow for the quick creation of portable, reliable networked applications.

The concept of a socket is based on a network model where various media exist with unique properties. Each of these media is described by the protocol it supports. An application, then, would have to establish a communications end point, or socket, to be connected to the desired medium. The destination application would, likewise, need another socket on the same medium.

The Sockets API provides an interface to setup, modify, and tear down these communications end points or sockets. Because the individual characteristics of the media are hidden from the application by the API, the underlying network can be changed without affecting the supported protocols. Similarly, as new protocols are added, application developers can use them using the same familiar interface they are already used to.

B. EXTERNAL INTERFACES

The table below illustrates the various function calls that define the Sockets API as implemented in PANSAT. The function signatures are identical to those used in the vast majority of the Sockets API implementations, so it should be fairly simple for anyone with sockets programming experience to write a sockets application for PANSAT.

Function	**Description**
socket	Opens a new socket and returns its handle.
bind	Assigns a socket address to a local socket.
listen	Enables a socket to accept incoming connections.
connect	Assigns a socket address for the remote socket.
accept	Waits for a listening socket to receive and establish a connection, returns a new connected socket.
send	Transmits a message over a connected socket.
sendto	Transmits a message over an unconnected socket.
recv	Waits for a message on a connected socket.
recvfrom	Waits for a message on an unconnected socket.
shutdown	Prepares a socket to be closed.
closesocket	Closes a socket permanently.

Table 6 - Sockets Interface

The functions represent a subset of the full functionality of the traditional implementations of the Sockets API. They were chosen for their consistency with the objectives of this development effort. Any function that was not deemed to have a high probability of use in a PANSAT application was discarded to keep the code size down to a minimum.

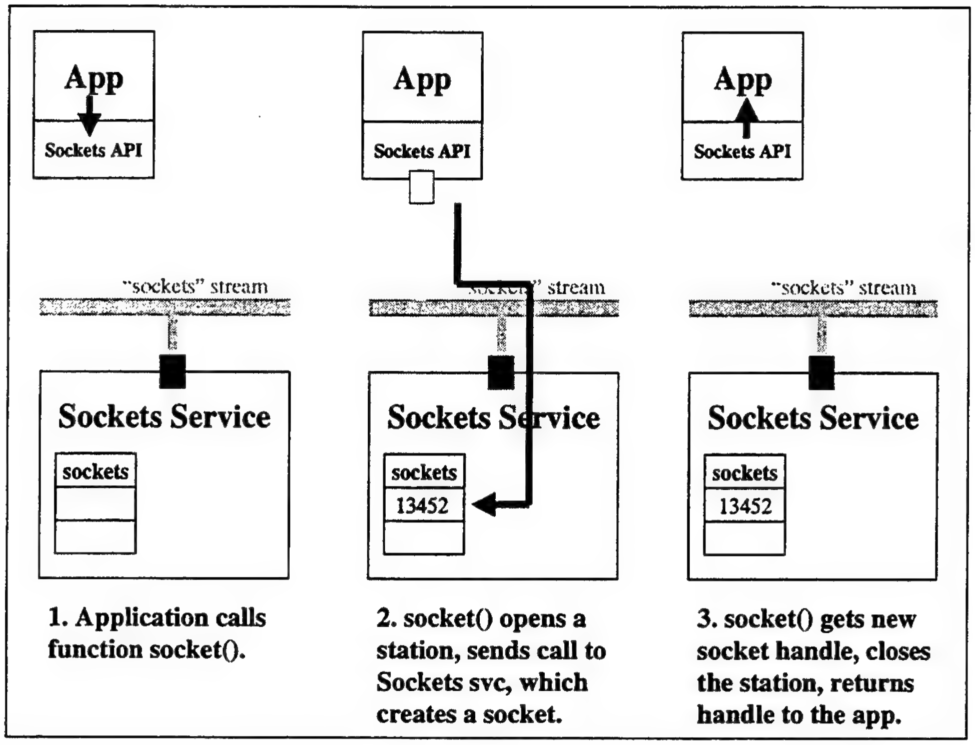
The following paragraphs describe in detail each of the functions that comprise the Sockets API in PANSAT. For the most part, their functionality should be identical to that of other Sockets implementations.

1. **socket()**

```
SOCKET socket( int af, int type, int protocol )
```

The socket() function opens a socket and returns a descriptor for it. The socket is opened when the data structure representing it has been allocated and initialized. The descriptor is a handle for the socket. The descriptor is a unique integer value used by the protocol stack to uniquely reference the socket.

The only allowed value for af (address family) in this implementation is PF_INET (protocol family: Internet). The allowed values for type are SOCK_DGRAM for a UDP socket and SOCK_RAW for a raw IP socket (such as would be needed for ICMP messages). Any other types result in **socket** returning an error code. Unless the socket type is SOCK_RAW, the protocol field should be set to zero, though this value is currently unused.



App
Sockets API
"sockets" stream
Sockets Service
sockets
13452
1. Application calls function socket().
2. socket() opens a station, sends call to Sockets svc, which creates a socket.
3. socket() gets new socket handle, closes the station, returns handle to the app.

Figure 6 - socket() Function Call

The **socket** function simply creates a station on the appropriate SCOS stream for the protocol, signals the appropriate Sockets service to create a default socket for this station, awaits confirmation from the protocol service, closes the station on the stream, and returns the handle for the newly created socket. The SCOS stream used by the Sockets API to exchange messages with the Sockets service is called "sockets." Note that each socket is uniquely associated with its corresponding protocol at the time it is created. This means that it is not possible to switch the protocol type of the socket without closing (destroying) it.

socket only sets the values of the socket handle and protocol. All other fields are reset to zero. This means that the address field for the socket is the generic value INADDR_ANY (which is actually 0.0.0.0). This address is invalid to receive datagrams, though it is allowed as the source address for an outgoing message. (The IP module replaces the INADDR_ANY value with the address of a real interface as it processes the datagram for transmission.)

The **socket** function returns a socket descriptor when it succeeds and the value INVALID_SOCKET when it fails. An attempt to open a socket would fail if no more memory was available or if the parameters provided were invalid. The call could also fail if the intended protocol service was not operating normally.

2. bind()

```
int bind( SOCKET s, struct sockaddr *addr, int namelen )
```

The **bind** function names the local socket with the address values in the sockaddr structure. **Bind** must be called on a server socket before it can accept client connections. This is required, because the **socket** function only initializes a generic socket, but does not assign it an address or port number. In order for the socket to send and receive messages, it must be uniquely identified by three values: address, port number, and protocol. The **socket** function only initializes the protocol value.

The bind() function returns zero on success and SOCKET_ERROR when it fails. bind() would fail if the socket parameter was not valid.

3. listen()

```
int listen( SOCKET s, int backlog )
```

The **listen** function prepares the server socket to accept connection requests from a client. Specifically, it sets the status flag on the socket to indicate that it can accept connection requests, and allocates a new connection queue for the socket. The value of the parameter backlog indicates the size of the connection queue.

The **listen** function returns zero on success or SOCKET_ERROR on failure. **listen** would fail if the socket provided was not valid or if the function was already called on this socket.

4. **connect()**

```
int connect( SOCKET s, struct sockaddr *addr, int namelen )
```

The **connect** function sets the destination socket address (address, port, protocol) for the socket provided. These values are stored in the socket structure. It is necessary to call the connect function before the **send** function can be invoked, but it is optional to do so before calling **sendto**.

If TCP sockets were supported, the **connect** function would establish an actual connection with the destination socket. However, since only connectionless protocols (UDP and Raw IP) are supported in this implementation of PANSAT, the **connect** function can be substantially simpler.

The **connect** function returns zero if it succeeds or SOCKET_ERROR if it fails. The function would fail if the socket referenced was invalid.

5. **accept()**

```
SOCKET accept( SOCKET s, struct sockaddr *addr, int *addrlen )
```

The **accept** function accepts a datagram and creates a new connected socket for this client. This allows the server to continue listening on the original socket for any new client requests. The newly created socket has a different name than the parent socket, and is intended as a temporary socket that will be closed when there is no more traffic for its client.

The **accept** function returns a descriptor to a new socket if it is successful or INVALID_SOCKET if it is not.

6. send()

```
int send( SOCKET s, const char *buf, int len, int flags )
```

The **send** function sends a data stream over the connection on which the socket is associated. There is no need to specify the address of the destination socket, since this was done as part of the call to **connect** (if the process is a client) or **accept** (if it is a server).

The **send** function returns the number of bytes sent on success and SOCKET_ERROR on failure. The function would fail if the socket was not named (result of calling **bind**) or if it was not associated (result of calling **connect**).

7. sendto()

```
int sendto( SOCKET s, const char *buf, int len, int flags,
            struct sockaddr *to, int tolen )
```

The **sendto** function differs from the **send** function only in that it allows a single socket to send datagrams to multiple destinations. If sendto() is called on a connected socket, the sockaddr provided in the function call is ignored and the remote socket address for the connection is used. In this situation, the address parameters of the function call are ignored.

The sendto() function returns the number of bytes sent on success and SOCKET_ERROR on failure. sendto() would fail if the socket parameter were invalid.

8. recv()

```
int recv( SOCKET s, char *buf, int len, int flags )
```

The recv() function waits for a message on a connected socket. The recv() function returns the number of bytes received if it is succesful and SOCKET_ERROR if it is not.

9. recvfrom()

```
int recvfrom( SOCKET s, char *buf, int len, int flags,
              struct sockaddr, int fromlen )
```

The recvfrom() function waits for a message on a socket. After the function succeeds, the sockaddr structure contains the socket address for the socket that sent the data. The recv() function returns the number of bytes received if it is succesful and SOCKET_ERROR if it is not.

10. closesocket()

```
int closesocket( SOCKET s )
```

The closesocket() function closes the connection on a connected (TCP) stream socket and returns the socket resources to the protocol stack. The closesocket() function returns zero on success and SOCKET_ERROR on failure.

11. shutdown()

```
int shutdown( SOCKET s, int how )
```

The shutdown() function performs a "partial close" of a connected socket so that closesocket() can be reliably called. This function is not required before closing a UDP or Raw IP socket. There are three ways to shutdown the connection. They are specified by the value of how (0 disallows receives, 1 disallows sends, and 2 means sends and receives are disallowed). The shutdown() function returns zero on success and SOCKET_ERROR on failure.

VIII.CONCLUSIONS

There are several problems that collectively thwart this effort to implement an IP protocol stack on PANSAT. Chief among them is the restriction against modifying the operating system kernel. Without being able to modify the OS and incorporate into it the services of the protocol stack, any implementation must be excessively complex and lengthy. This effect complicates the problems that were faced when trying to deal with the limited amount of memory resources available.

The problems with getting enough memory to operate the complex machinery of these protocols had serious and rippling effects throughout this effort. By default, SCOS is tailored to support processes with small memory requirements. This is a fair approach, as most microsatellites have several simple tasks executing concurrently. In this case, however, the small memory model default required changes to the entire structure of the programs.

Though SCOS supports medium and large memory models for more complex tasks, the supporting files for these models were not available for this effort. As a result, even after tinkering with the stack limits, the programs had to have an unusually flat structure. Function calls that were nested four or layers deep in the program sometimes led to stack overflows. Any level of nesting beyond that would consistently cause this problem. This phenomenom caused significant additional effort, since the programs were originally designed in a modular way which made extensive use of nested function calls.

In summary, if an IP stack is to be developed to run in user mode on SCOS, it is imperative to acquire the files required to implement the large memory model within that OS. Even if this were to be accomplished, the amount of memory required to implement and operate the protocol stack, when combined with the memory required by the kernel and its tasks, would account for much of the available memory on the satellite. This seems to indicate that SCOS may not be well suited for this application.

LIST OF REFERENCES

BekTek. *BekTek Spacecraft Operating System Reference Manual.* December 1992.

BekTek. *BekTek AX.25 Reference Manual.* December 1992.

Braden, R., "Requirements for Internet Hosts -- Communication Layers," RFC-1122, October 1989.

Braden, R., "Requirements for Internet Hosts -- Application and Support," RFC-1123, October 1989.

Postel, J., "Internet Control Message Protocol (ICMP)," RFC-792, September 1981.

Postel, J., "Internet Protocol (IP)," RFC-791, September 1981

Postel, J., "Transmission Control Protocol," RFC-793, September 1981.

Postel, J., "User Datagram Protocol," RFC-768, August 1980.

Quinn, B. and Shute, D., *Windows Sockets Network Programming*, Addison-Wesley, 1996

Stallings, W., *Data and Computer Communications*, Prentice Hall, 1997.

Stevens, W.R., *TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley*, 1994.

Stevens, W. R. and Wright, G. R., *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1995.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. INTERNET PROTOCOL REQUIREMENTS

1. Implement IP and ICMP
2. Silently discard datagrams whose version is not 4
3. Verify IP checksum and silently discard bad datagram
4. Addressing:
    - 4.1. Subnet addressing compliant with RFC-950
    - 4.2. Source address must be host's own IP address
    - 4.3. Silently discard datagram with bad destination address
    - 4.4. Silently discard datagram with bad source address
5. Support reassembly
6. Allow transport layer to set type of service (TOS)
7. Time-to-Live (TTL):
    - 7.1. Must not send packet with TTL of 0
    - 7.2. Must not discard received packets with TTL < 2
    - 7.3. Allow transport layer to set TTL
    - 7.4. Fixed TTL is configurable
8. IP Options:
    - 8.1. Allow transport layer to send IP options
    - 8.2. Pass all IP options received to higher layer
    - 8.3. IP layer silently ignores unknown options
    - 8.4. Silently ignore Stream Identifer option
9. Source Route (SR) Option:

9.1. Originate & terminate Source Route options

9.2. Datagram with completed source route passed up to transport layer

9.3. Build correct (non-redundant) return route

9.4. Must not send multiple source route options in one header

10. ICMP:

10.1. Silently discard ICMP messages with unknown type

10.2. Demultiplex ICMP Error to transport protocol

10.3. Must not send ICMP error message for:

10.3.1. ICMP error message

10.3.2. IP broadcast or IP multicast

10.3.3. Link-layer broadcast

10.3.4. Non-initial fragment

10.3.5. Datagram with non-unique source address

10.4. Return ICMP error messages (when not prohibited)

10.5. Pass ICMP "Destination Unreachable" to higher layer

10.6. Redirect:

10.6.1. Update route cache when receiving Redirect

10.6.2. Handle both Host and Net Redirects

10.7. Pass Source Quench to higher layer

10.8. Time Exceeded: pass to higher layer

10.9. Pass Parameter Problem to higher layer

10.10. ICMP Echo Request or Reply:

10.10.1. Echo server

10.10.2. Use specific-destination address as Echo Reply source

10.10.3. Send same data in Echo Reply

10.10.4. Pass Echo Reply to higher layer

10.10.5. Reverse and reflect Source Route option

10.11. ICMP Address Mask Request and Reply:

10.11.1. Address Mask source configurable

10.11.2. Support static configuration of address mask

11. ROUTING OUTBOUND DATAGRAMS:

11.1. Use address mask in local/remote decision

11.2. Operate with no gateways on conn network

11.3. Maintain "route cache" of next-hop gateways

11.4. If no cache entry, use default gateway

11.4.1. Support multiple default gateways

11.5. Able to detect failure of next-hop gateway

11.6. Switch from failed default gateway to another

11.7. Manual method of entering configuration information

12. REASSEMBLY and FRAGMENTATION:

12.1. Able to reassemble incoming datagrams

12.1.1. At least 576 byte datagrams

12.2. Transport layer able to learn the Maximum Message Size that can be Received (MMS_R)

12.3. Send ICMP Time Exceeded on reassembly timeout

12.4. Pass Maximum Message Size that can be Sent (MMS_S) to higher layers

12.5. Don't send datagrams bigger than MMS_S

13. MULTIHOMING:

13.1. Allow application to choose local IP address

14. BROADCAST:

14.1. Recognize all broadcast address formats

14.2. Use IP broadcast/multicast address in link-layer broadcast

15. INTERFACE:

15.1. Allow transport layer to use all IP mechanisms

15.2. Pass interface identification up to transport layer

15.3. Pass all IP options up to transport layer

15.4. Transport layer can send certain ICMP messages

15.5. Pass specified ICMP messages up to transport layer

15.5.1. Include IP hdr+8 octets or more from original datatram

APPENDIX B. IP INTERFACE MODULE SOURCE CODE

```
/**********************************************************************/
/* Project     : A Sockets API for PANSAT                             */
/* File        : ip_interface.h                                       */
/* Author      : Fernando J. Maymi                                    */
/* Date        : 30 May 2000                                          */
/* Description: This file identifies the data structures that         */
/*    describe the interfaces used by IP on PANSAT.                   */
/**********************************************************************/

#ifndef IP_INTERFACE_H
#define IP_INTERFACE_H

#include "qax25.h"

#define FRAME_SIZE 256
#define QUEUE_SIZE 2


/* Structure describing an interface used by IP over BAX
*/
struct ip_if {
   unsigned long address;
   unsigned long netmask;
   char in_q[QUEUE_SIZE][ FRAME_SIZE ];
   char out_q[QUEUE_SIZE][ FRAME_SIZE ];
   short in_q_length;
   short out_q_length;
   struct ip_if far *next_if;
   struct ip_if far *prev_if;
   struct CONTROL_BLOCK cb;
   unsigned int channel;
};

#endif
```

```
/**********************************************************************/
/* Project     : A Sockets API for PANSAT                             */
/* File        : ip_if.c                                              */
/* Author      : Fernando J. Maymi                                    */
/* Date        : 30 May 2000                                          */
/* Description: This file implements the functions that manipulate   */
/*    the IP interfaces on PANSAT.                                    */
/**********************************************************************/

#include <stdlib.h>
#include "ip_if.h"
#include "ip.h"
#include "qax25.h"
#include "kernal.h"


/* The attach function creates a new interface structure and         */
/* initializes all fields. If the control block value is not null,   */
/* the function will also register the interface with the BAX task   */
/* so that it may connect. Lastly, the function prepends the         */
/* interface to the list of interfaces pointed by the list parameter.*/


struct ip_if far *if_attach( unsigned long addr, unsigned long nmsk,
         struct ip_if far *list )
{
   static unsigned char id = '0';    /* qualifies interface callsign*/

   int i, error = 0;
   char far *temp_ptr;
   struct ip_if far *new_if;

   new_if = (struct ip_if far *)fmalloc( sizeof( struct ip_if ) );

   if ( new_if ) {                    /* struct allocation ok      */

      /* Initialize the fields in the interface structure.         */

      new_if->address = addr;
      new_if->netmask = nmsk;
      new_if->in_q_length = 0;
      new_if->out_q_length = 0;
      new_if->next_if = list;
      new_if->prev_if = 0;
        new_if->channel = 0;

      /* Set the callsign for this interface & attempt to claim it */
      /* in the BAX task. Notice that each PANSAT AX.25 interface  */
      /* has a callsign of the form PANIPx where x is an integer   */
      /* value in the range 0-9. This allows other non-IP services */
      /* or programs to use the PANSAT callsign, and the IP stack  */
      /* can support up to 10 distinct network interfaces. Note    */
      /* the loopback interface does not need/claim a BAX callsign.*/

      if ( nmsk != LOOPBACK_MASK ) {

         qax_clean_cb( &(new_if->cb) );
```

```
            strcpy( new_if->cb.my_call.call, "PANIP" );
            if ( id < '9' ) {
               new_if->cb.my_call.call[5] = id++;
               new_if->cb.my_call.ssid = 0;
            }

            /* When claiming the callsign, a return value of 0 shows */
            /* success, negative value indicates callsign already in */
            /* use and a positive value indicates another BAX error. */

            error = qax_claim( &(new_if->cb), 0 );

            /* If the error code is not zero, the interface couldn't */
            /* claim a callsign and is removed.                       */

            if ( error ) {

               new_if->next_if = 0;
               ffree( new_if );
               new_if = list;          /* return ptr to orig. list   */
            }
         }
      }

      else                                 /* couldn't allocate new i/f */
         new_if = list;                   /* return ptr to orig. list  */

      return new_if;
   }



   /* The detach function removes the indicated interface from the list */
   /* of interfaces. The return value is 0 if it's succesful, or else 1.*/

   struct ip_if far *if_detach( struct ip_if far *this_if,
                        struct ip_if far *list )
   {
      struct ip_if far *new_if_list = list;

      if ( this_if && list ) {                 /* avoid null pointers    */
         if ( list == this_if )               /* this is 1st interface */
            new_if_list = this_if->next_if; /* point to nxt interface*/
         else {
            new_if_list = list;
            this_if->prev_if->next_if = this_if->next_if;
         }
         if ( this_if->next_if )              /* this isn't last one    */
            this_if->next_if->prev_if = this_if->prev_if;
      }

      return new_if_list;
   }



   /* The disconnect function terminates the interface's BAX connection.*/
```

```
/* If the function is succesful, it returns a zero, otherwise a 1.   */

int if_disconnect( struct ip_if far *this_if )
{
   int result = 0;

   result = qax_disconnect( this_if->cb.channel, 0 );

   return result;
}



/* if_find returns a pointer to the interface that matches the        */
/* address and/or netmask within the provided list of interfaces. If */
/* either the address or netmask argument is 0, that argument is not */
/* considered in the search. If both are 0, or if no match is found, */
/* it returns zero.                                                   */

struct ip_if far *if_find( unsigned long addr, unsigned long nmsk,
                   struct ip_if far *list )
{
   struct ip_if far *result = 0;

   if ( addr ) {                            /* if address is not 0   */
      result = list;
      if ( nmsk ) {                         /* and netmask is not 0  */
         while ( result ) {
            if ( ( result->address == addr ) &&
                 ( result->netmask == nmsk ) )
               break;                    /* interface found       */
            else
               result = result->next_if;
         }
      }
      else {                                /* match addr, not nmsk  */
         while ( result ) {
            if ( result->address == addr ) /* only check address */
               break;                       /* interface found    */
            else
               result = result->next_if;  /* check next         */
         }
      }
   }
   else {                                   /* addr is 0, check nmsk */
      if ( nmsk ) {                         /* nmsk is not 0         */
         result = list;
         while ( result ) {
            if ( result->netmask == nmsk ) /* only check address */
               break;                       /* interface found    */
            else
               result = result->next_if;
         }
      }
   }
   return result;
}
```

```
/* The transmit function attempts to transmit the message. If the    */
/* outgoing queue for this interface is not empty, those messages    */
/* already in the queue are transmitted first. If the interface is   */
/* not able to transmit immediately, the message is placed in the    */
/* outgoing queue and the length of the queue is returned. If the    */
/* dgram pointer is zero, if_send will attempt to flush all pending  */
/* messages on the outgoing queue. If the message(s) is(are)         */
/* transmitted, 0 is returned.                                       */

int if_send( struct ip_if far *this_if, char *dgram )
{
   int ix = 0;
   int error = 0;
   struct ip_header far *ip;

   /* If this is the loopback interface, check for room on incoming */
   /* queue. If there is, add message to this interface's incoming  */
   /* queue.                                                        */

   if ( this_if->address & LOOPBACK_MASK ) {
      if ( this_if->in_q_length < QUEUE_SIZE ) {

         charcopy( dgram, this_if->in_q[ this_if->in_q_length ],
            0, 0, FRAME_SIZE );
         this_if->in_q_length++;

      }
      else                                         /* queue full   */
         error = this_if->in_q_length + 1;        /* return error */

      return error;
   }

   /* Before attempting to send message, check the queue of pending */
   /* outgoing messages. If messages already waiting, try to send   */
   /* them first.                                                   */

   while ( ( this_if->out_q_length > 0 ) && !error ) {
      ip = (struct ip_header far *)this_if->out_q[ 0 ];

      /* try to send the dgram: fail if queue full or no connect   */

      if ( !qax_data_full( this_if->cb.channel, ip->length ) )
         error = qax_data( 0, 0, this_if->out_q[ 0 ], ip->length );

      if ( !error ) {
         /* these for loops shift the pointers in the input queue */
         for ( ix = 0; ix < ( this_if->out_q_length - 1 ); ix++ )
            strcpy( this_if->out_q[ ix ],
                    this_if->out_q[ ix + 1 ], FRAME_SIZE);
         this_if->out_q_length--;
      }
   } /* end while loop */
   /* If this interface is not the loopback, ensure there is enough */
   /* room to send message. If there is, send it. Otherwise, if     */
   /* there is room on interface's outgoing queue, put the message  */
   /* there and return the length of the outgoing queue.            */

   if ( !error ) {

      ip = (struct ip_header far *)dgram;
      if ( ~qax_data_full( this_if->cb.channel, ip->length ) ) {
         error = qax_data( 0, 0, dgram, ip->length );
      }
      else {
         if ( this_if->out_q_length < QUEUE_SIZE )
            strcpy( this_if->out_q[ this_if->out_q_length++ ],
               dgram, FRAME_SIZE );
         else
            error = this_if->out_q_length + 1;
      }

   } /* end if ( !error ) */

   return error;
}


/* The receive function checks for pending incoming messages. If any */
/* are found, they are added to the interface's incoming queue. The  */
/* function removes the last message from the queue and returns a    */
/* pointer to it. If there are no messages queued, 0 is returned.    */

char far *if_receive( struct ip_if far *this_if )
{
   char msg[80];
   char far *dgram = 0;
   int ix, length;

   dgram = fmalloc( FRAME_SIZE );

   /* If this is the loopback interface, return the next datagram.  */

   if ( this_if->address & LOOPBACK_MASK ) {

      if ( this_if->in_q_length > 0 ) {
         _farmemcpy( dgram, this_if->in_q[ this_if->in_q_length ],
                     FRAME_SIZE );
         this_if->in_q_length--;

        }
      else {                                            /* queue empty */
         ffree( dgram );
         dgram = 0;
      }
   }

   /* Check the connection status. Attempt to connect if a client   */
   /* has tried to establish a connection. Otherwise, the interface */
   /* can't receive.                                                 */
```

```
    else {
       while ( qax_input_ready( -1 ) &&          /* message from BAX */
          ( this_if->in_q_length < QUEUE_SIZE ) ) {

          qax_input( -1, &(this_if->cb), dgram, &length );

          switch ( this_if->cb.type ) {
          case QAT_DATA:
             _farmemcpy( this_if->in_q[ this_if->in_q_length++ ],
                         dgram, FRAME_SIZE );
             break;
          case QAT_STATE:
             break;
          case QAT_UI:
             break;
          } /* end of switch statement */
       }

       /* Now all pending messages from BAX have been processed,    */
       /* thandle he last message in this interface's input queue.  */

       if ( this_if->in_q_length > 0 ) {

          _farmemcpy( dgram, this_if->in_q[ this_if->in_q_length ],
                      FRAME_SIZE );
          this_if->in_q_length--;

       }
       else {
          ffree( dgram );
          dgram = 0;
       }

    }

    return dgram;

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: IP MODULE SOURCE CODE

```
/**********************************************************************/
/* Project     : A Sockets API for PANSAT                              */
/* File        : ip.h                                                  */
/* Author      : Fernando J. Maymi                                     */
/* Date        : 09 May 2000                                           */
/* Description: This file identifies the data and functions that      */
/*    tdescribe he external interfaces to IP on PANSAT.                */
/**********************************************************************/

#ifndef IP_H
#define IP_H


#define MTU                         1024
#define REASSEMBLY_TIMEOUT            30

#define BAD_DST_MASK                0x000000E0
#define LOOPBACK_MASK               0x0000007F

#define DEFAULT_TTL                         25


/* These definitions describe stream and message characteristics      */

#define BUFF_CNT       50           /* number of buffers         */
#define BUFF_SIZE     256           /* size of each buffer       */


/* Definitions for the values of the protocol field */
#define ICMP 1
#define RAW 4
#define TCP 6
#define UDP 17


/* Structure of an internet header without any options               */
struct ip_header {

   unsigned char  header_length:4,         /* header length         */
            version:4;                    /* ip version            */
   unsigned char  tos;                     /* type of service       */
   short          length;                  /* total length          */
   unsigned short id;                      /* datagram id           */
   short          offset;                  /* fragment offset       */
#define IP_DF 0x4000                        /* don't fragment flag   */
#define IP_MF 0x2000                        /* more fragments flag   */
#define IP_OFFMASK 0x1FFF                   /* mask for fragmenting  */
   unsigned char  ttl;                     /* time to live          */
   unsigned char  protocol;                /* higher-layer protocol */
   unsigned short checksum;                /* header checksum       */

   unsigned long  source,                  /* source address        */
           destination;                   /* destination address   */
                              /* (both in net byte order)  */
};


/* Structure for the head of a list of fragments belonging to a dgram*/
struct frag_list {
   long   timestamp;                      /* arrival of 1st frag   */
   short  tail_recvd;                     /* arrival of last frag  */
   struct ip_header far *dgram_header;    /* IP header of datagram */
   struct frag far *next_frag;            /* next fragment on list */
   struct frag_list far *next_list;       /* next list of frags    */
   struct frag_list far *prev_list;       /* prev list of frags    */
};


/* Structure of a datagram fragment                                     */
struct frag {
   short  start;                           /* 1st data byte offset */
   short  end;                             /* last data byte offset*/
   char  far *data;                        /* ptr to this fragment */
   struct frag far *next_frag;             /* next frag on list    */
   struct frag far *prev_frag;             /* previous frag on list*/
};


/* The charcopy function copies the number of characters indicated    */
/* by the len field from src at (starting) offset src_off to dst at  */
/* (starting) offset dst_off.                                         */

void charcopy( char *src, char *dst, short src_off, short dst_off,
          short len );


/* The cksum function performs an Internet checksum on the data       */
/* pointed to by msg. The length of the message is indicated by len. */
/* This function is called by both the IP and transport layers.       */

unsigned short cksum( char *msg, int len );

#endif
```

```
/*********************************************************************/
/* Project     : A Sockets API for PANSAT                            */
/* File        : ip.c                                                */
/* Author      : Fernando J. Maymi                                   */
/* Description: This file implements the Internet Protocol for       */
/*   PANSAT. It elies on the services provided by BekTek's SCOS and  */
/*   BAX, specially for the link layer.                              */
/*********************************************************************/


#include <stdlib.h>
#include "qax25.h"
#include "qcferr.h"
#include "kernal.h"
#include "ip.h"
#include "ip_icmp.h"
#include "ip_if.h"



/* Global variable declarations */
int ip_idcounter = 1;
struct frag_list far *fragged_dgrams = 0;  /* dgram frag lists       */
struct ip_if far *ip_interfaces = 0;       /* list of interfaces     */



/*-------------------------------------------------------------------*/
/* The charcopy function copies each data byte from the source to    */
/* the destination, starting at offset start, and ending at offset   */
/* end minus one.                                                    */

void charcopy( char *src, char *dst, short src_off, short dst_off,
           short len )
{
   char *src_ptr = src + src_off;
   char *dst_ptr = dst + dst_off;
   short i;

   for ( i = 0; i < len; i++ ) {
      *(dst_ptr + i)= *(src_ptr + i);
   }
}



/*-------------------------------------------------------------------*/
/* The cksum function is copied almost verbatim from Wright &        */
/* Stevens book "TCP/IP Illustrated, Volume 2".                      */

unsigned short cksum( char *ip, int len )
{
   long sum = 0;

   while ( len > 1 ) {
      sum += *( (unsigned short *) ip )++;
```

```
        if ( sum & 0x80000000 )
           sum = ( sum & 0xFFFF ) + ( sum >> 16 );
        len -= 2;
    }

    if ( len )
        sum += (unsigned short) *(unsigned char *) ip;

    while ( sum >> 16 )
        sum = ( sum & 0xFFFF ) + ( sum >> 16 );

    return ~(unsigned short)sum;

}



/*-------------------------------------------------------------------*/
/* ip_net_addresss_valid tests a given network address for validity. */
/* It returns zero if the address is not valid.                      */

int ip_net_address_valid( long net_address )
{
    int result;

    if ( ( net_address & BAD_DST_MASK ) == BAD_DST_MASK )
        result = 0;
    else
        result = 1;

    return result;
}



/*-------------------------------------------------------------------*/
/* ip_free_fraglist frees all memory allocated to a fragment list.   */
/* This includes the frag_list and frag structures and the pointers  */
/* to the char data.                                                 */

void ip_free_fraglist( struct frag_list far *lp )
{
    struct frag far *fragment, *next_fragment;

    /* Prune this fragment list from the list of fragment lists      */
    if ( fragged_dgrams == lp )                  /* first frag list */
        fragged_dgrams = lp->next_list;
    else                                         /* not first list  */
        lp->prev_list->next_list = lp->next_list;
    if ( lp->next_list ) {                       /* any more lists? */
        lp->next_list->prev_list = lp->prev_list;
        lp->next_list = 0;
    }
    lp->prev_list = 0;

    /* Free all resources associated with this list of fragments     */
    fragment = lp->next_frag;
    while ( fragment ) {                   /* remove all frags from list*/
       next_fragment = fragment->next_frag;
       fragment->next_frag = 0;
       fragment->prev_frag = 0;
       ffree( fragment->data );
       fragment->data = 0;
       ffree( fragment );

       fragment = next_fragment;  /* point to next frag to be freed */
    }

    /* All frags have been freed up, so free the list header now     */
    ffree( lp->dgram_header );
    lp->dgram_header = 0;
    ffree( lp );
    lp = 0;
 }



 /*-----------------------------------------------------------------*/
 /* ip_reassemble receives a fragment of a datagram. It then looks    */
 /* for more fragments of the same datagram and attempts to           */
 /* reassemble it. If it succeeds, it passes the reassembled datagram */
 /* to the calling process for delivery. Otherwise, it stores the     */
 /* fragment in the appropriate list.                                 */

 int ip_reassemble( char *dgram )
 {
    char   *ptr;
    int    result = 0;                  /* dgram been reassembled?   */
    short  length = 0;                  /* length of a block of data */
    short  src_offset,                  /* source and dest offsets   */
           dst_offset;
    struct frag_list far *lp = fragged_dgrams;  /* to lists of frags */
    struct frag far *fp = 0;            /* pointer to a frag in list */
    struct frag far *new_frag;          /* pointer to the new frag   */

    struct ip_header *header =
       (struct ip_header *)dgram;  /* overlay ip header on fragment */

    /* Create a new fragment structure to hold the fragment's payload*/
    new_frag = (struct frag far *)fmalloc( sizeof( struct frag ) );
    new_frag->start = header->offset;
    new_frag->end = header->offset +
               ( header->length - (short)header->header_length );
    new_frag->data = (char far *)fmalloc
       ( new_frag->end - new_frag->start );
    length = header->length - header->header_length;
    ptr = dgram + header->header_length;
    _farmemcpy( new_frag->data, ptr, length );
    new_frag->next_frag = 0;
    new_frag->prev_frag = 0;

    /* Use lp as a running pointer to the lists of fragments. For     */
    /* each fragment list on main list, look for 4-way match between */
    /* id, source, destination, and protocol of the new fragment and */
```

```
/* the fragments already on the list. If a match is found, set lp*/
/* to point to list of fragments that match received fragment.   */
while ( (int)lp ) {
   if ( ( lp->dgram_header->id == header->id )
       && (lp->dgram_header->source == header->source)
       && (lp->dgram_header->destination == header->destination)
       && (lp->dgram_header->protocol == header->protocol) )
      break;
   else
      lp = lp->next_list;
}

if ( lp ) {                          /* more frags for this dgram */

   fp = lp->next_frag;              /* first frag on list        */

   if ( fp->start > new_frag->start ) { /* new head of list frag*/
      new_frag->next_frag = fp;
      fp->prev_frag = new_frag;
      lp->next_frag = new_frag;
   }

   else {                                  /* find right spot    */
      while ( ( fp->next_frag ) &&        /* next frag not null*/
              ( fp->end < new_frag->start ) )
         fp = fp->next_frag;              /* point to next frag*/

      /* once out of the while loop, the process has either     */
      /* found a fragment with an offset greater than that of  */
      /* the new frag, or it's at the end of the frag list.    */

      if ( ~(int)fp ) {              /* end of list: append     */
         fp->next_frag = new_frag;
         new_frag->prev_frag = fp;
      }

      else {                         /* not end: insert         */
         new_frag->next_frag = fp;
         new_frag->prev_frag = fp->prev_frag;
         fp->prev_frag->next_frag = new_frag;
         fp->prev_frag = new_frag;
      }

   }

   if ( header->offset & ~IP_MF )      /* last frag of dgram     */
      lp->tail_recvd = 1;             /* tag as having the tail*/

   /* once the new fragment has been inserted into the frag list*/
   /* the process checks to see if all fragments have arrived.  */
   /* If so, the function reassembles the original datagram and */
   /* returns it to the calling process.                        */

   fp = lp->next_frag;                 /* point fp to first frag*/

   /* ensure the first and last fragments have been received    */
   if ( ( fp->start == 0 ) && ( lp->tail_recvd ) ){
        /* now, ensure there are no gaps on the list of fragments*/
        while ( fp ) {
           if ( ( fp->next_frag != 0 ) &&
                ( fp->end + 1 == fp->next_frag->start ) ) {
              fp = fp->next_frag;     /* no gap: go to next frag*/
              length = fp->next_frag->end;     /* track size    */
           }
        }

        /* if there are no gaps, and all tests are ok reassemble */
        if ( fp == 0 ) {

           /* copy original dgram header                        */
           length = lp->dgram_header->header_length * 4;
           _farmemcpy( dgram, lp->dgram_header, length );

           /* update header fields that are now obsolete        */
           header = (struct ip_header *)dgram;
           header->checksum = 0;
           header->length = sizeof( struct ip_header );

           /* copy all fragments into the datagram, in order    */
           ptr = dgram + header->length; /* point to begin data */
           fp = lp->next_frag;
           while ( fp != 0 ) {
              length = fp->end - fp->start + 1;
              _farmemcpy( ptr, fp->data, length );
              ptr += length;
              fp = fp->next_frag;
           }

           /* free up the fragment list                         */
           ip_free_fraglist( lp );

           result = 1;
        }
     }
  }

  else {                                    /* fragment of new dgram */
     lp = (struct frag_list far *)fmalloc
        ( sizeof( struct frag_list ) );
     lp->timestamp = time( 0 );
     length = header->header_length * 4;
     lp->dgram_header =
        (struct ip_header far *)fmalloc( length );
     strcpy( lp->dgram_header, dgram, length );
     lp->next_frag = new_frag;
     lp->next_list = fragged_dgrams;
     lp->prev_list = 0;
     if ( header->offset & ~IP_MF )  /* last frag of orig. dgram */
        lp->tail_recvd = 1;         /* tag list as having tail   */
     else
        lp->tail_recvd = 0;
     fragged_dgrams = lp;
  }
```

```
   /* examine all fragment lists to ensure none are stale          */
   lp = fragged_dgrams;
   while ( lp ) {
      if ( ( lp->timestamp + REASSEMBLY_TIMEOUT ) > time( 0 ) )
         ip_free_fraglist( lp );
   }

   return result;
}


/*-----------------------------------------------------------------*/
/* The ip_output function is called by main if there is a message   */
/* for the ip process in the "ip" stream. The function processes the */
/* message and, if it conforms to the IP RFC, transmits it on the   */
/* appropriate interface.                                           */

int ip_output( char *dgram )
{
   int good_dgram = 1;
   int error = 0;
   unsigned long netmask = 0;
   struct ip_header *ip;
   struct ip_if far *outgoing_if;

   /* Overlay ip header                                             */
   ip = (struct ip_header *)dgram;

   /* Initialize header fields that were not set by the transport   */
   /* layer. Note that raw IP packets have all fields already set.  */
   if ( ip->protocol != RAW ) {
      ip->version = 4;
      ip->offset = 0;
      ip->id = ip_idcounter++;
   }

   if ( ip->header_length < 5 )
      good_dgram *= 0;
    if ( ip->ttl < 1 )
      good_dgram *= 0;
    if ( !ip_net_address_valid( ip->destination ) )
      good_dgram *= 0;

   if ( !good_dgram ) {                      /* bad dgram: discard */
      error = 1;
   }

   else {                                    /* good dgram: send it */

      /* If the source address is specified, but invalid, or it is */
      /* not specified at all, assign it the address of the first  */
      /* local interface.                                          */

      if ( ip->source ) {               /* check valid source addr */
         outgoing_if = if_find( ip->source, netmask,
            ip_interfaces );
         if ( outgoing_if == 0 ) {
            outgoing_if = ip_interfaces;
            ip->source = outgoing_if->address;
         }
      }

      else {                           /* assign first valid source addr */
         outgoing_if = ip_interfaces;
         ip->source = outgoing_if->address;
      }

      ip->checksum = cksum( dgram, ip->header_length );

      /* Drop outgoing datagrams that exceed the max transmission  */
      /* unit (MTU). This check is separate from other validity    */
      /* checks to facilitate the implementation of fragmentation. */

      if ( ip->length > MTU ) {             /* discard dgram          */
         error = 1;
      }

      /* The datagram is ready for transmission, so transmit it.   */
      if ( dgram )
         error = if_send( outgoing_if, dgram );
   }

   return error;
}


/*-----------------------------------------------------------------*/
/* The main body of ip initializes the ip service and enters a loop  */
/* until a fatal error is encountered.                                */

void main()
{
   char temp_buf[ BUFF_SIZE ];
   int len = 0;                          /* length of dgram buffer    */
   int error = 0;
   unsigned int ip_pool;
   unsigned long address,
            netmask;
   struct MD ip_md;
   struct SCB ip_scb;                    /* station control block     */

   /* variable declarations for input processing */
   char *dgram;
   char far *ip_buf;
   int good_dgram;                       /* datagram valid flag       */
   int complete_dgram;                   /* dgram not missing frags   */
   char icmp_code = 0,                   /* for generating ICMP msgs  */
      icmp_type = 0;
   unsigned long icmp_param = 0;
   struct ip_header *ip;
   struct ip_if far *this_if = ip_interfaces;
```

```
/* Initialize ip.                                                      */

/* attach the loopback interface */
address = 0x0100007F;
netmask = 0x0000007F;
ip_interfaces = if_attach( address, netmask, ip_interfaces );
/* attach a notional interface with an IP address: 44.136.8.5 and*/
/* net mask of 255.255.255.0                                         */
address = 0x0508882C;
netmask = 0x00FFFFFF;
ip_interfaces = if_attach( address, netmask, ip_interfaces );
address = 0;
netmask = 0;

len = 0;
this_if = ip_interfaces;
while (this_if) {
   len++;
   sprintf( msg, "Interface %d at %d (%d / %d).\n", len, this_if,
      this_if->prev_if, this_if->next_if );
   _qcf_print( msg );
   this_if = this_if->next_if;
}

/* open station ip on stream ip, creating the stream if necessary*/
 qcf_default_scb( &ip_scb, "ip", "ip" );
ip_scb.max_write = 50;
error += qcf_open( &ip_scb, "wf" );
error += qcf_build_pool("ip", "c", BUFF_CNT, BUFF_SIZE, &ip_pool);

while ( !error ) {

   /* check "ip" stream for messages from transport protocols.  */
   /* If error is detected, the loop's broken and ip shuts down.*/

   if ( !qcf_read( &ip_scb, &ip_buf, 0, NULL, 0, ERR ) ) {

      _farmemcpy( temp_buf, ip_buf, ip_scb.v.bufsize );
      qcf_rel_pool_buf( ip_buf );
      ip_output( temp_buf );
   }


   /*************************************************************/
   /* This part of ip main program checks for/processes input.  */
   /* Inputs come from the BAX task or from the output function */
   /* if the interface is the loopback.                          */
   /*-----------------------------------------------------------*/

   this_if = ip_interfaces;

   /* Check interfaces for pending dgrams in incoming queues.   */

   while( this_if ) {          /* while there's a valid interface */

      dgram = if_receive( this_if );
```

```
while( dgram != 0 ) { /* while there are dgrams in queue */
   good_dgram = 1;
   complete_dgram = 1;
   ip = (struct ip_header *)dgram;

   /* Verify version, hdr len and cksum of the incoming */
   /* dgram. Also check the source and destination addr */
   /* A failed check causes the dgram to be marked bad. */

   if ( ip->version != 4 )
      good_dgram *= 0;
   if ( ip->header_length == 5 )
      good_dgram *= 0;
   if ( ip->checksum !=
       cksum( dgram, ip->header_length ) )
      good_dgram *= 0;
   if ( !ip_net_address_valid( ip->source ) )
      good_dgram *= 0;
   if ( if_find( ip->destination, netmask,
       ip_interfaces ) == 0 ) {
      good_dgram *= 0;
      icmp_type = ICMP_UNREACH;
      icmp_code = ICMP_UNREACH_HOST_UNKNOWN;
   }

   if ( good_dgram )
      if ( ip->offset & ~IP_DF )
         if ( ~ip_reassemble( dgram ) )
            complete_dgram = 0;

   /* If the dgram is valid and complete, IP passes it  */
   /* to the appropriate transport layer.               */

   if ( good_dgram && complete_dgram ) {

      switch ( ip->protocol ) {
      case ICMP:
         ip_icmp_input( dgram );
         dgram = 0;
         break;

      case RAW: case UDP:
         ip_buf = qcf_get_pool_buf( ip_pool, BLOCK );
         _farmemcpy( ip_buf, dgram, ip->length );
         strcpy( ip_md.stn_name, "sockets" );
         qcf_write( &ip_scb, ip_buf, ip->length, NULL,
            NOWAIT );
         dgram = 0;
         break;

      default:
         icmp_code = ICMP_UNREACH;
         icmp_type = ICMP_UNREACH_PROTOCOL;
         ip_icmp_output( dgram, icmp_type, icmp_code,
            icmp_param );
         dgram = 0;
```

```
                  break;
               }
            }

            else {                                /* dgram bad: discard */

               if ( icmp_code ) {
                  ip_icmp_output( dgram, icmp_type, icmp_code,
                     icmp_param );
                  icmp_code = 0;
                  icmp_type = 0;
                  icmp_param = 0;
               }
               dgram = 0;
            }

            dgram = if_receive( this_if );  /* get next datagram */

         } /* end of datagram "while" loop */

         this_if = this_if->next_if;         /* check next i/f    */

      } /* end of interface "while" loop */

   } /* end of the "while( 1 )" loop */

   /* If execution gets to this point, a major error has ocurred &  */
   /* the IP service will shut down.                                */

   qcf_close( &ip_scb, PURGE );

} /* end of main() */
```

APPENDIX D. SOCKETS MODULE SOURCE CODE

```
/**********************************************************************/
/* Project:     A Sockets API for PANSAT                              */
/* File:        sockets.h                                             */
/* Author:      Fernando J. Maymi                                     */
/* Date:        26 May 2000                                           */
/* Description: This file defines the codes and structures used      */
/*    internally by the sockets API and the transport protocols.      */
/**********************************************************************/

#ifndef SOCKETS_H
#define SOCKETS_H

#include "sockapi.h"

/* The following codes define the messages that can be exchanged      */
/* between the sockets API and the transport services.                */

#define SOCKET                  1
#define BIND                    2
#define LISTEN                  3
#define CONNECT                 4
#define ACCEPT                  5
#define GETSOCKOPT              6
#define SETSOCKOPT              7
#define SEND                    8
#define SENDTO                  9
#define RECV                    10
#define RECVFROM                11
#define CLOSESOCKET             12
#define SHUTDOWN                13


/* The following definitions describe streams and messages            */

#define BUFF_CNT                50      /* number of buffers          */
#ifndef BUFF_SIZE
#define BUFF_SIZE               256     /* size of each buffer        */
#endif


/* Definitions for the socket states and queue size                   */

#define DISCONNECTED             0
#define CONNECTING               1
#define CONNECTED                2
#define WAITING_ACCEPT           3      /* waiting for new connect    */
#define WAITING_RECV             4      /* waiting for a datagram     */
#define WAITING_RECVFROM         5      /* waiting for a datagram     */


#define QUEUE_SIZE              16      /* max msgs in socket queues */
```

```
/* The sock_msg structure is used to encode the type of socket system*/
/* call and the return value of the system call and the socket handle*/

struct sock_msg_hdr {
   int      sock_id;                     /* socket handle for call    */
   short    sys_call;                    /* desired system call       */
   short    rtn_code;                    /* return value              */
};


/* The following structures are overlays for the data portion of a   */
/* SCOS stream message containing a socket system call.                */

struct param_socket {                     /* Parameter list for Socket */
   int af,                               /* address family (AF_INET)  */
      type,                             /* type (i.e. SOCK_DGRAM)    */
      protocol;                         /* protocol (not used)       */
};


struct param_dgram {                      /* Parameters for Send/Recv  */
   struct sockaddr_in address;           /* socket address (src/dst)  */
   int flags;                            /* flags for socket options  */
   int len;                              /* length of the data        */
   char far *data;                       /* data                      */
};


/* The socket structure maintains all information regarding a given   */
/* socket, including its input and output queues.                      */

struct socket {
   int               handle;
   unsigned long     sock_address;
   unsigned short    sock_port;
   unsigned long     dest_address;
   unsigned short    dest_port;
   short             flags;
   short             state;
    short             type;                 /* generic type     */
    short             options;              /* from socket call */
   char              stn_name[8];          /* for blocking calls */
   struct socket * next_socket;
   struct socket * prev_socket;
};

/* The udp_header structure is a standard header for a UDP datagram   */

struct udp_header {
   unsigned short source,
                    destination,
                 length,
                 checksum;
};
#endif
```

```
/*********************************************************************/
/* Project:      A Sockets API for PANSAT                            */
/* File:         sockets.c                                           */
/* Author:       Fernando J. Maymi                                   */
/* Date:         09 June 2000                                        */
/* Description: This file provides the implementation of the Sockets */
/*    API system calls defined in sockets.h                          */
/*********************************************************************/

#include <stdlib.h>
#include "ip.h"
#include "kernal.h"
#include "sockapi.h"
#include "sockets.h"




/*-------------------------------------------------------------------*/
/* The cksum function is copied almost verbatim from Wright & Stevens*/
/* book "TCP/IP Illustrated, Volume 2".                              */

unsigned short udp_cksum( char *msg, int len )
{
   long sum = 0;

   while ( len > 1 ) {
      sum += *( (unsigned short *) msg )++;
      if ( sum & 0x80000000 )
         sum = ( sum & 0xFFFF ) + ( sum >> 16 );
      len -= 2;
   }

   if ( len )
      sum += (unsigned short) *(unsigned char *) msg;

   while ( sum >> 16 )
      sum = ( sum & 0xFFFF ) + ( sum >> 16 );

   return ~(unsigned short)sum;

}



/*-------------------------------------------------------------------*/
/* The clear_socket option resets all values of a socket structure.  */
/* clear_socket is called by main() when a socket struct is created. */

void clear_socket( struct socket *the_socket )
{
   int ix;

   the_socket->sock_address = 0;
   the_socket->sock_port = 0;
   the_socket->dest_address = 0;
   the_socket->dest_port = 0;
   the_socket->next_socket = 0;
   the_socket->prev_socket = 0;
}



/*------------------------------------------------------------------*/
/* The find_socket function returns a pointer to the socket handle   */
/* matching the value of the argument. find_socket is called when a  */
/* socket system call message, except SOCKET, is received.           */

struct socket *find_socket( int handle, struct socket *list )
{
   struct socket *this_socket = list;

   while( this_socket ) {
      if ( this_socket->handle == handle )
         break;
      else
         this_socket = this_socket->next_socket;
   }

   return this_socket;
}



/*------------------------------------------------------------------*/
/* The find_sockaddr function finds the socket (if it exists) that   */
/* matches the socket address parameter. It returns a pointer to the */
/* socket if it exists, and zero otherwise.                          */

struct socket *find_sockaddr( struct sockaddr_in *addr,
                              struct socket *list )
{
   char msg[80];
   unsigned long this_addr = addr->sin_addr;
   unsigned long this_port = addr->sin_port;
   struct socket *that_socket = list;

   /* Search through list of sockets until there are no more. If a  */
   /* match is found, break the loop and return the pointer to it.  */
   while ( that_socket ) {
      if ( ( this_addr == that_socket->sock_address ) &&
         ( this_port == that_socket->sock_port ) ) {
         break;
      }
      else
         that_socket = that_socket->next_socket;
   }

   return that_socket;                    /* return socket or zero */

}
```

```
void main()
{
   char temp_buf1[ 80 ],                       /* scratchpad for msgs    */
        temp_buf2[ 80 ];
   char *temp_buf_data;               /* ptr to temp_buf data   */
   int empty = 0;
   int error = 0;
   int next_handle = 42;
   struct socket *socket_list = 0;         /* ptr to sockets list    */

   /* declarations pertaining to SCOS streams (sockets, ip)         */
   char far *buffer = 0;                       /* memory buffers         */
   char far *ip_buffer;
   int len;                                    /* length of buffers      */
   int sock_pool,                              /* pool numbers           */
      ip_pool;
   struct MD sock_md;                          /* msg descriptor ptrs    */
   struct MD ip_md;
   struct SCB sock_scb,                        /* station ctrl blocks    */
            ip_scb;

   /* declarations of structures to overlay on received messages    */
   struct ip_header *ip;
   struct param_dgram *dgram_parameter;
   struct param_socket *sock_parameter;
   struct sock_msg_hdr *message;
   struct socket *this_socket,
              *that_socket;
   struct sockaddr_in *address;
   struct udp_header *udp;

   /* open station sockets on stream ip                             */
   strcpy( ip_md.stn_name, "ip" );
    qcf_default_scb( &ip_scb, "ip", "sockets" );
   ip_scb.max_write = 50;
   error += qcf_open( &ip_scb, "rf" );
   error += qcf_build_pool("ip", "r", BUFF_CNT, BUFF_SIZE, &ip_pool);

   /* open station on stream sockets, creating stream if needed     */

    qcf_default_scb( &sock_scb, "sockets", "sockets" );
   sock_scb.max_write = 50;
   error += qcf_open( &sock_scb, "wf" );
   error += qcf_build_pool( "sockets", "w", BUFF_CNT, BUFF_SIZE,
      &sock_pool );

   while ( !error ) {

      /* check the "sockets" stream for messages from the API.     */
      empty = qcf_read( &sock_scb, &buffer, len, &sock_md, 1, ERR );

      if( !empty ) {
         /* process the message */
         empty = 0;
         this_socket = 0;
```

```
/* copy sockets buffer contents to local char buffer for */
/* easy access, then overlay the sock_msg_hdr structure  */
_farmemcpy( temp_buf1, buffer, sock_scb.v.bufsize );
message = (struct sock_msg_hdr *)temp_buf1;

switch ( message->sys_call ) {

case SOCKET:
   sock_parameter = (struct param_socket *)(temp_buf1
      + sizeof(struct sock_msg_hdr) );
   if ( sock_parameter->af != AF_INET ) {
      message->rtn_code = 0;
      break;            /* bad address family, abort */
   }

   this_socket =
      (struct socket *)fmalloc( sizeof(struct socket) );

   /* ensure fmalloc succeeded and sock family is valid */
   if ( this_socket ) {
      clear_socket( this_socket );
      this_socket->handle = next_handle++;
      this_socket->type = sock_parameter->type;
      this_socket->state = DISCONNECTED;

      /* now, insert the socket at head of the list    */
      this_socket->next_socket = socket_list;
      this_socket->next_socket->prev_socket =
         this_socket;
      socket_list = this_socket;

      /* if all works, return the socket handle (id)   */
      message->rtn_code = this_socket->handle;
   }
   else                              /* fmalloc failed    */
      message->rtn_code = 0;      /* signal an error   */
   break;

case BIND:
   this_socket = find_socket
      ( message->sock_id, socket_list );

   /* if no socket with provided handle exists, abort.  */
   if ( !this_socket ) {
      message->rtn_code = ENOTSOCK;
      break;
   }

   /* Search through list of sockets to ensure no other */
   /* socket already uses the requested socket address. */
   address = (struct sockaddr_in *)( temp_buf1
      + sizeof( struct sock_msg_hdr ) );
   that_socket = socket_list;
   while ( that_socket ) {
      if ( (address->sin_addr ==
            that_socket->sock_address)
         && (address->sin_port ==
                 that_socket->sock_port)
              && (this_socket->type == that_socket->type) )
              break;
           else
              that_socket = that_socket->next_socket;
        }

        /* If this_socket is not 0, socket with the requested*/
        /* address already exists. Set error code and break. */
        if ( that_socket ) {
           message->rtn_code = EADDRINUSE;
           break;
        }

        /* Everything checks out, so bind the socket.         */
        this_socket->sock_address = address->sin_addr;
        this_socket->sock_port = address->sin_port;
        message->rtn_code = 0;
        break;


     case LISTEN:
        this_socket = find_socket( message->sock_id,
           socket_list );
        if ( this_socket ) {
           this_socket->flags = SO_ACCEPTCONN;
           message->rtn_code = 0;
        }
        else
           message->rtn_code = ENOTSOCK;
        break;


     case CONNECT:
        this_socket = find_socket( message->sock_id,
           socket_list );
        if ( this_socket ) {
           address = (struct sockaddr_in *)( message
              + sizeof( struct sock_msg_hdr ) );
           this_socket->dest_address = address->sin_addr;
           this_socket->dest_port = address->sin_port;
           this_socket->state = CONNECTED;
           message->rtn_code = 0;
        }
        else
           message->rtn_code = 1;       /* signal an error    */
        break;


     case ACCEPT:
        /* if matching socket exists and it's had BIND called*/
        /* all that's needed is to set its state to WAIT and */
        /* save the socket's station name so that a message  */
        /* can be sent to it when a datagram arrives.        */
        this_socket = find_socket( message->sock_id,
           socket_list );
        if ( !this_socket ) {
```

```
      message->rtn_code = ENOTSOCK;
      break;
   }
   if ( this_socket->flags != SO_ACCEPTCONN ) {
      message->rtn_code = EOPNOTSUPP;
      break;
   }
   this_socket->state = WAITING_ACCEPT;
   strcpy( this_socket->stn_name,
      sock_scb.v.res_md.stn_name );
   break;

case SEND:
   this_socket = find_socket( message->sock_id,
      socket_list );
   if ( !this_socket ) {
      message->rtn_code = ENOTSOCK;
      break;
   }
   if ( this_socket->state != CONNECTED ) {
      message->rtn_code = ENOTCONN;
      break;
   }

   len = message->rtn_code;
   strcpy( ip_md.stn_name, "ip" );

   switch ( this_socket->type ) {
   case SOCK_DGRAM:
      /* build a default (incomplete) ip header        */
      ip = (struct ip_header *)temp_buf2;
      ip->header_length = 5;
      ip->length = len + sizeof( struct ip_header )
         + sizeof( struct udp_header );
      ip->tos = 0;
      ip->ttl = DEFAULT_TTL;
      ip->protocol = UDP;
      ip->source = this_socket->sock_address;
      ip->destination = this_socket->dest_address;

      /* build the udp header                          */
      udp = (struct udp_header *)
         (temp_buf2 + sizeof(struct ip_header) );
      udp->source = this_socket->sock_port;
      udp->destination = this_socket->dest_port;
      udp->length = ip->length - ip->header_length;
      udp->checksum = udp_cksum( (char *)udp,
         udp->length );

      /* copy the payload and transmit                 */
      temp_buf_data = (char *)(udp + sizeof
         (struct udp_header));
      strcpy( temp_buf_data, temp_buf1,
         message->rtn_code );
      ip_buffer = qcf_get_pool_buf( ip_pool, BLOCK );
      if ( !ip_buffer ) {
         message->rtn_code = 55;
```

```
            break;
         }
         _farmemcpy( ip_buffer, temp_buf2, ip->length );
         message->rtn_code = qcf_write( &ip_scb, ip_buffer,
            ip->length, &ip_md, NOWAIT );
         break;

      case SOCK_RAW:
         ip_buffer = qcf_get_pool_buf( ip_pool, BLOCK );
         len = message->rtn_code;
         _farmemcpy( ip_buffer, temp_buf1, len );
         message->rtn_code = qcf_write( &ip_scb, ip_buffer,
            len, &ip_md, NOWAIT );
         break;

      default:
         message->rtn_code = EPROTONOSUPPORT;
         break;

      } /* end of nested switch( this_socket->type ) */
      break;


   case RECV:
      /* If the socket is found, set its state to indicate */
      /* it is waiting to complete the RECV call; save the */
      /* socket station name so that a message can be sent */
      /* to it when a datagram is received.                */
      this_socket = find_socket( message->sock_id,
         socket_list );
      if ( this_socket ) {
         this_socket->state = WAITING_RECV;
         strcpy( this_socket->stn_name,
            sock_scb.v.res_md.stn_name );
      }
      else
         message->rtn_code = ENOTSOCK;
      break;


   case RECVFROM:
      /* If the socket is found, set its state to indicate */
      /* it is waiting to complete the RECVFROM; save the  */
      /* socket station name so that a message can be sent */
      /* to it when a datagram is received.                */
      this_socket = find_socket( message->sock_id,
         socket_list );
      if ( this_socket ) {
         this_socket->state = WAITING_RECVFROM;
         strcpy( this_socket->stn_name,
            sock_scb.v.res_md.stn_name );
      }
      else
         message->rtn_code = ENOTSOCK;
      break;
```

```
case CLOSESOCKET:
   this_socket = find_socket( message->sock_id,
      socket_list );
   if ( this_socket ) {

      /* socket is first in list, point list to next   */
      if ( this_socket == socket_list ) {
         socket_list = this_socket->next_socket;
      }
      else {

         /* socket not first, bypass it, then delete  */
         this_socket->prev_socket->next_socket =
            this_socket->next_socket;

         /* ensure this is not last socket in the list*/
         if ( this_socket->next_socket )
            this_socket->next_socket->prev_socket =
               this_socket->prev_socket;
      }
      message->rtn_code = 0;
   }
   else
      message->rtn_code = ENOTSOCK;
   break;


case SHUTDOWN:
   /* not implemented */
   message->rtn_code = 0;
   break;


default:
   message->rtn_code = 1;
   break;
} /* end of switch statement */


/* Send response to application with the return code.    */
/* This message is an echo of the original message, but  */
/* with return code value set. If the socket is waiting  */
/* for a datagram (after calling accept, recv, etc.) do  */
/* not reply, release the pool buffer instead. This will */
/* cause socket to block until the datagram is received. */

if ( this_socket->state < WAITING_ACCEPT ) {
   strcpy( sock_md.stn_name, sock_scb.v.res_md.stn_name );
   len = sock_scb.v.bufsize;
   _farmemcpy( buffer, temp_buf1, len );
   qcf_write( &sock_scb, buffer, len, &sock_md, NOWAIT );

   /* See if the socket needs to be deleted */
   if ( message->sys_call == CLOSESOCKET &&
       message->rtn_code != ENOTSOCK ) {
      clear_socket( this_socket );
      ffree( this_socket );
```

```
            }
        }
        else
            qcf_rel_pool_buf( buffer );

    } /* end of if statement (msg from socket layer) */

    /* check the "ip" stream for messages from the ip service    */
    empty = qcf_read( &ip_scb, &buffer, len, &ip_md, 1, ERR );
    if( !empty ) {
        /* process the message */
        empty = 0;
        this_socket = 0;

        /* copy sockets buffer contents to local char buffer for */
        /* easy access, then overlay the sock_msg_hdr structure  */
        _farmemcpy( temp_buf2, buffer, ip_scb.v.bufsize );
        qcf_rel_pool_buf( buffer );
        ip = (struct ip_header *)temp_buf2;

        switch ( ip->protocol ) {

        case RAW:
            /* each waiting Raw socket gets copy of each Raw msg */
            this_socket = socket_list;
            while ( this_socket ) {

                /* examine each socket to determine if it is Raw */
                /* and waiting for a datagram.                   */
                if ( ( this_socket->type == SOCK_RAW ) &&
                         ( (this_socket->state == WAITING_RECV) ||
                      (this_socket->state == WAITING_RECVFROM) ||
                      (this_socket->state == WAITING_ACCEPT) ) ) {

                    /* make a copy of the message */
                    strcpy( sock_md.stn_name,
                        this_socket->stn_name );
                    buffer = qcf_get_pool_buf( sock_pool, BLOCK );
                    _farmemcpy( buffer, temp_buf2, ip->length );
                    qcf_write( &sock_scb, buffer, ip->length,
                        &sock_md, NOWAIT );
                }

                /* point to next socket to be examined           */
                this_socket = this_socket->next_socket;
            }
            break;

        case UDP:
            /* overlay udp header on the right offset within the */
            /* dgram to retrieve the destination socket port     */
            udp = (struct udp_header *)
                (temp_buf2 + sizeof( struct ip_header ) );

            /* to determine the recepient socket, a sockaddr     */
            /* structure is needed so that the service can call  */
            /* find_sockaddr().                                  */
            address = (struct sockaddr_in *)
               fmalloc( sizeof( struct sockaddr_in ) );

            /* use the ip and udp header information to fill the */
            /* values in temporary sockaddr_in structure and call*/
            /* find_sockaddr().                                  */
            address->sin_addr = ip->destination;
            address->sin_port = udp->destination;
            this_socket = find_sockaddr( address, socket_list );

            /* if socket was found, check if it is waiting for a */
            /* datagram. If it is, send it immediately. If not,  */
            /* silently discard the datagram.                    */
            if ( this_socket ) {

               if ( ( this_socket->state == WAITING_ACCEPT ) ||
                    ( this_socket->state == WAITING_RECV ) ||
                    ( this_socket->state == WAITING_RECVFROM ) ) {

                  /* socket was waiting, finish the reception  */
                  strcpy( sock_md.stn_name,
                     this_socket->stn_name );
                  buffer = qcf_get_pool_buf( sock_pool, BLOCK );
                  _farmemcpy( buffer, temp_buf2, len );
                  qcf_write( &sock_scb, buffer, len, &sock_md,
                     NOWAIT );

                  /* reception completed, clear the wait flag  */
                  if ( this_socket->state == WAITING_RECV )
                     this_socket->state = CONNECTED;
                  else
                     this_socket->state = DISCONNECTED;
               }

               /* if the socket was not waiting for a datagram, */
               /* it is discarded as a security precaution      */

            }

            ffree( address );
            break;

         default:
            break;

      } /* end of switch(ip->protocol) */

   } /* end of if statement (msg from ip layer) */

} /* end while( !error ) */

/* If execution ever gets to this point, a major error has        */
/* ocurred and the UDP service will shut down.                    */
qcf_close( &sock_scb, PURGE );
qcf_close( &ip_scb, PURGE );

}
```

APPENDIX E. SOCKETS API SOURCE CODE

```
/*********************************************************************/
/* Project:      A Sockets API for PANSAT                            */
/* File:         sockapi.h                                           */
/* Author:       Fernando J. Maymi                                   */
/* Date:         09 May 2000                                         */
/* Description: This file contains the specification for the sockets */
/*    application programming interface (API) developed by the       */
/*    author for the Petite Amateur Naval Satellite (PANSAT) at the  */
/*    U.S. Naval Post-Graduate School in Monterey, California.       */
/*********************************************************************/


#ifndef SOCKAPI_H
#define SOCKAPI_H


/* address family */
#define AF_INET                 2

/* socket types */
#define SOCK_DGRAM              2
#define SOCK_RAW                3

/* socket option flags                                               */
#define SO_ACCEPTCONN           0x0002

/* socket states                                                     */
#define SS_NOFDREF              0x0001
#define SS_ISCONNECTED          0x0002
#define SS_ISCONNECTING         0x0004
#define SS_ISDISCONNECTING      0x0008
#define SS_CANTSENDMORE         0x0010
#define SS_CANTRCVMORE          0x0020

#define SS_NBIO                 0x0100
#define SS_ASYNC                0x0200
#define SS_ISCONFIRMING         0x0400

#define SS_INCOMP               0x0800
#define SS_COMP                 0x1000
#define SS_ISDISCONNECTED       0x2000

/* socket errors
*/
#define ENOTSOCK                38
#define EDESTADDRREQ            39
#define EMSGSIZE                40
#define EPROTOTYPE              41
#define ENOPROTOOPT             42
#define EPROTONOSUPPORT         43
#define ESOCKTNOSUPPORT         44
#define EOPNOTSUPP              45
#define EPFNOSUPPORT            46
```

```
#define EADDRINUSE            48
#define EADDRNOTAVAIL         49
#define ENETDOWN              50
#define ENETUNREACH           51
#define ENETRESET             52
#define ECONNABORTED          53
#define ECONNRESET            54
#define ENOBUFS               55
#define EISCONN               56
#define ENOTCONN              57
#define ESHUTDOWN             58
#define ETIMEDOUT             60
#define ECONNREFUSED          61

/* SCOS stream message types for each function call                  */
#define SM_SOCKET              0
#define SM_BIND                1
#define SM_LISTEN              2
#define SM_CONNECT             3
#define SM_ACCEPT              4
#define SM_GETSOCKOPT          5
#define SM_SETSOCKOPT          6
#define SM_SEND                7
#define SM_SENDTO              8
#define SM_RECV                9
#define SM_RECVFROM           10
#define SM_CLOSESOCKET        11
#define SM_SHUTDOWN           12


struct sockaddr {
   unsigned char sa_len;          /* total length              */
   unsigned char sa_family;       /* addr family: AF_INET      */
   char          sa_data[14];     /* address value             */
};


struct sockaddr_in {
   unsigned char  sin_len;        /* always 16 for PANSAT (IP) */
   unsigned char  sin_family;     /* always AF_INET for PANSAT */
   unsigned short sin_port;       /* port #, net byte order    */
   unsigned long  sin_addr;       /* IP addr, net byte order   */
   char           sin_zero[8];    /* filler to match sockaddr  */
};


int socket( int af, int type, int protocol );
   /* parameters: af is the protocol suite, type is the protocol    */
   /* type, and protocol is the protocol name.                      */
   /* returns: new socket descriptor.                               */


int bind( int socket, struct sockaddr *addr, int namelen );
   /* parameters: s is an unbound socket, addr is the local port    */
   /* number and IP address, and namelen is the length of the addr  */
   /* structure.                                                    */
   /* returns: 0 on success or error code on failure.               */
```

```
int listen( int socket, int backlog );
   /* parameters: s is a named, unconnected socket, and backlog is  */
   /* the desired length of the incoming queue.                     */
   /* returns: 0 on success or error code on failure.               */


int connect( int socket, struct sockaddr *addr, int namelen );
   /* parameters: s is unconnected socket, addr is the remote port  */
   /* and IP address, and namelen is the length of the addr struct. */
   /* returns: 0 on success or error code on failure.               */


int accept( int socket, struct sockaddr *addr, int *addrlen );
   /* parameters: s is a listening socket, addr is the socket addr  */
   /* for the new socket created by the accept function, and        */
   /* addrlen is the length of the addr structure.                  */
   /* returns: a descriptor for a new (connected) socket            */


int send( int socket, const char *buf, int len, int flags );
   /* parameters: s is a connected socket, buf is a pointer to the  */
   /* buffer containing outgoing data, len is number of bytes to    */
   /* send, and flags are the option flags for the socket.          */
   /* returns: 0 on success or error code on failure.               */


int sendto( int socket, const char *buf, int len, int flags,
   struct sockaddr *to, int tolen );
   /* parameters: s is valid socket, buf is pointer to the buffer   */
   /* containing the outgoing data, len is number of bytes to send, */
   /* flags is option flag value for the socket, to is a pointer to */
   /* the structure containing the remote socket address, and tolen */
   /* is the length of the to structure.                            */
   /* returns: 0 on success or error code on failure.               */


int recv( int socket, char *buf, int len, int flags );
   /* parameters: s is connected socket, buf is pointer to buffer   */
   /* for the incoming data, len is  number of bytes received, and  */
   /* flags is the value of the socket flags.                       */
   /* returns: 0 on success or error code on failure.               */


int recvfrom( int socket, char *buf, int len, int flags,
   struct sockaddr *from, int fromlen );
   /* parameters: s is valid socket, buf is pointer to buffer for   */
   /* the incoming data, len is number of bytes received, flags is  */
   /* value of the socket flags, from is pointer to the structure   */
   /* containing remote socket address, and fromlen is length of    */
   /* the from structure.                                           */
   /* returns: 0 on success or error code on failure.               */


int closesocket( int socket );
   /* parameter: s is a valid socket.                               */
   /* returns: 0 on success or error code on failure.               */


int shutdown( int socket, int how );
   /* parameter: s is a valid socket, and how is a flag describing  */
   /* the desired shutdown behavior.                                */
   /* returns: 0 on success or error code on failure.               */

#endif
```

```
/************************************************************************/
/* Project:     A Sockets API for PANSAT                                */
/* File:        sockapi.c                                               */
/* Author:      Fernando J. Maymi                                       */
/* Date:        09 May 2000                                             */
/* Description: This file contains the implementation of the sockets */
/*    application programming interface (API) developed by the author*/
/*    for the Petite Amateur Naval Satellite (PANSAT) at the U.S.     */
/*    Naval Post-Graduate School in Monterey, California.            */
/************************************************************************/


#include <stdlib.h>
#include "kernal.h"
#include "sockapi.h"
#include "sockets.h"



void sock_call( char *msg, int *len )
{
   char far *buffer;
   static int pool;
   int sock_error = 0,
      scos_error = 0;
   static struct SCB sock_scb;
   struct MD sock_md;


   /* Initialize the SCB for this call on the sockets stream. The   */
   /* name of the station is set to NULL so that SCOS assigns a     */
   /* nunique station ame automatically. Also get the mSG buffers.  */

   qcf_default_scb( &sock_scb, "sockets", NULL );
   sock_scb.max_write = 1;
   scos_error += qcf_open( &sock_scb, "rf" );
   scos_error += qcf_build_pool("sockets", "r", 1, BUFF_SIZE, &pool);

   /* If the station opens successfully and the buffers pool exists,*/
   /* send a socket message to the socket service station indicated */
   /* in the "switch" statement. Otherwise, report a socket error.  */

   if ( !scos_error ) {

      /* get stream msg buffer and copy the scratchpad data to it  */
      buffer = qcf_get_pool_buf( pool, BLOCK );
      _farmemcpy( buffer, msg, *len );

      /* set the destination station name and send the message     */
      strcpy( sock_md.stn_name, "sockets" );
      qcf_write( &sock_scb, buffer, *len, &sock_md, NOWAIT );

      /* After sending message, wait (indefinitely) for response   */
      qcf_read( &sock_scb, &buffer, 0, &sock_md, 0, NOERR );

      /* After receiving the response, set return value and return.*/
      *len = sock_scb.v.bufsize;
```

```
        _farmemcpy( msg, buffer, *len );

    }

    qcf_rel_pool_buf( buffer );
    qcf_rel_pool( pool );
    qcf_close( &sock_scb, PURGE );
}


int socket( int af, int type, int protocol )
{
    char temp_buf[ BUFF_SIZE ];          /* local "scratchpad"         */
    int len;                             /* length of buffer           */
    int sock_error = 0;
    struct sock_msg_hdr *message;        /* socket message header      */
    struct param_socket *params;         /* parameters for socket call*/

    /* overlay sockets msg header on the local buffer and set values */
    message = (struct sock_msg_hdr *)temp_buf;
    message->sock_id = 0;
    message->sys_call = SOCKET;
    message->rtn_code = 0;

    /* place the call parameters in the data portion of the message  */
    params = (struct param_socket *)
          (temp_buf + sizeof(struct sock_msg_hdr));
    params->af = af;
    params->type = type;
    params->protocol = protocol;

    len = sizeof( struct sock_msg_hdr ) + sizeof( struct param_socket );
    sock_call( temp_buf, &len );
    message = (struct sock_msg_hdr *)temp_buf;
    sock_error = message->rtn_code;

    return sock_error;  /* return socket handle or error code if bad */

}


int bind( int socket, struct sockaddr *addr, int namelen )
{
    char temp_buf[ BUFF_SIZE ];          /* local "scratchpad"         */
    int len = 0;                         /* length of buffer           */
    int sock_error = 0;
    struct sock_msg_hdr *message;        /* socket message descriptor */
    struct sockaddr *addr_parameter;

    /* overlay sockets msg header on the local buffer and set values */
    message = (struct sock_msg_hdr *)temp_buf;
    message->sock_id = socket;
    message->sys_call = BIND;
    message->rtn_code = 0;

    /* overlay sockaddr structure on local buffer and copy the values*/
    addr_parameter = (struct sockaddr *)
```

```
      (temp_buf + sizeof( struct sock_msg_hdr ) );
   addr_parameter->sa_len = addr->sa_len;
   addr_parameter->sa_family = addr->sa_family;
   for ( len = 0; len < 14; len++ )
      addr_parameter->sa_data[ len ] = addr->sa_data[ len ];

   len = sizeof( struct sock_msg_hdr ) + sizeof( struct sockaddr );
   sock_call( temp_buf, &len );
   message = (struct sock_msg_hdr *)temp_buf;
   sock_error = message->rtn_code;

   return sock_error;  /* return socket handle or error code if bad */
}


int listen( int socket, int backlog )
{
   char temp_buf[ BUFF_SIZE ];          /* local "scratchpad"          */
   int len;                             /* length of buffer            */
   int sock_error = 0;
   struct sock_msg_hdr *message;        /* socket message descriptor */


   /* overlay sockets msg header on local buffer and set values     */
   message = (struct sock_msg_hdr *)temp_buf;
   message->sock_id = socket;
   message->sys_call = LISTEN;
   message->rtn_code = backlog;  /* use this field for parameter  */

   len = sizeof( struct sock_msg_hdr ) + sizeof( int );
   sock_call( temp_buf, &len );
   message = (struct sock_msg_hdr *)temp_buf;
   sock_error = message->rtn_code;

   return sock_error;  /* return socket handle or error code if bad */
}


int connect( int socket, struct sockaddr *addr, int namelen )
{
   char temp_buf[ BUFF_SIZE ];          /* local "scratchpad"          */
   int len;                             /* length of buffer            */
   int sock_error = 0;
   struct sock_msg_hdr *message;        /* socket message descriptor */
   struct sockaddr *addr_parameter;

   /* overlay sockets msg header on local buffer and set values     */
   message = (struct sock_msg_hdr *)temp_buf;
   message->sock_id = socket;
   message->sys_call = CONNECT;
   message->rtn_code = namelen;  /* use this field for parameter  */

   /* overlay sockaddr structure on the local buffer and copy values*/
   addr_parameter = (struct sockaddr *)
      (temp_buf + sizeof( struct sock_msg_hdr ) );
   addr_parameter->sa_len = addr->sa_len;
   addr_parameter->sa_family = addr->sa_family;
```

```
    for ( len = 0; len < 14; len++ )
       addr_parameter->sa_data[ len ] = addr->sa_data[ len ];

    len = sizeof( struct sock_msg_hdr ) + sizeof( struct sockaddr );
    sock_call( temp_buf, &len );
    message = (struct sock_msg_hdr *)temp_buf;
    sock_error = message->rtn_code;

    return sock_error;  /* return socket handle or error code if bad */
}


int accept( int socket, struct sockaddr *addr, int *addrlen )
{
    char temp_buf[ BUFF_SIZE ];          /* local "scratchpad"          */
    int len;                             /* length of buffer            */
    int sock_error = 0;
    struct sock_msg_hdr *message;        /* socket message descriptor */
    struct sockaddr *addr_parameter;


    /* overlay sockets msg header on the local buffer and set values */
    message = (struct sock_msg_hdr *)temp_buf;
    message->sock_id = socket;
    message->sys_call = ACCEPT;
    message->rtn_code = 0;

    len = sizeof( struct sock_msg_hdr ) + sizeof( struct sockaddr );
    sock_call( temp_buf, &len );
    message = (struct sock_msg_hdr *)temp_buf;
    sock_error = message->rtn_code;
sprintf( msg, "API: Accept: rtn_code: %d.\n", message->rtn_code );
_qcf_print( msg );

    return sock_error;  /* return socket handle or error code if bad */
}


int send( int socket, const char *buf, int len, int flags )
{
    char *ptr;
    char temp_buf[ BUFF_SIZE ];          /* local "scratchpad"          */
    int sock_error = 0;
    struct sock_msg_hdr *message;


    if ( len > BUFF_SIZE - sizeof( struct sock_msg_hdr ) )
       sock_error = 55;              /* ENOBUFFS */

    else {
       /* overlay sockets msg header on local buffer and set values */
       message = (struct sock_msg_hdr *)temp_buf;
       message->sock_id = socket;
       message->sys_call = SEND;
       message->rtn_code = len;      /* cheat: send param here!   */

       ptr = temp_buf + sizeof( struct sock_msg_hdr );
```

```
      strcpy( ptr, buf, len );
      len += sizeof( struct sock_msg_hdr );
      sock_call( temp_buf, &len );
      message = (struct sock_msg_hdr *)temp_buf;
      sock_error = message->rtn_code;
   }

   return sock_error;  /* return socket handle or error code if bad */
}


int sendto( int socket, const char *buf, int len, int flags,
   struct sockaddr *to, int tolen )
{
   char temp_buf[ BUFF_SIZE ];         /* local "scratchpad"         */
   char far *buffer;                   /* memory buffer              */
   int sock_error = 0;
   struct sock_msg_hdr *message;

   /* not implemented */
   return 0;
}


int closesocket( int socket )
{
   char msg[80];
   char temp_buf[ BUFF_SIZE ];         /* local "scratchpad"         */
   int len;
   int sock_error = 0;
   struct sock_msg_hdr *message;


   /* overlay sockets msg header on local buffer and set values     */
   message = (struct sock_msg_hdr *)temp_buf;
   message->sock_id = socket;
   message->sys_call = CLOSESOCKET;
   message->rtn_code = 0;  /* use this field for parameter    */

   len = sizeof( struct sock_msg_hdr );
   sock_call( temp_buf, &len );
   message = (struct sock_msg_hdr *)temp_buf;
   sock_error = message->rtn_code;
sprintf( msg, "API: CloseSocket: rtn_code: %d.\n", message->rtn_code );
_qcf_print( msg );

   return sock_error;  /* return socket handle or error code if bad */
}


int shutdown( int socket, int how )
{
   return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center................ 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library................................. 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, California 93943-5101

3. Gilbert M. Lundy.................................... 1
   Naval Postgraduate School
   Mail Code CS/LN
   542A Spanagel Hall
   833 Dyer Road
   Monterey, CA 93943-5000

4. Chairman, Department of Computer Science............ 1
   Naval Postgraduate School
   Spanagel Hall
   833 Dyer Road
   Monterey, CA 93943-5000

5. Fernando J. Maymi................................... 1
   2 University Circle
   SGC 2157
   Monterey, California 93943